DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF A COMPILER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE PHI - Φ

by

Eugene J. Cole
and
Joseph E. Connell II

June 1987

Thesis Advisor: Daniel Davis

Approved for public release; distribution is unlimited

T233190

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION
Unclassified

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION / DOWNGRADING SCHEDULE

3 DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
Distribution is Unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Naval Postgraduate School

6b OFFICE SYMBOL (If applicable)
Code 52

7a NAME OF MONITORING ORGANIZATION
Naval Postgraduate School

6c ADDRESS (City, State, and ZIP Code)
Monterey, California 93943-5000

7b ADDRESS (City, State, and ZIP Code)
Monterey, California 93943-5000

8a NAME OF FUNDING / SPONSORING ORGANIZATION

8b OFFICE SYMBOL (If applicable)

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c ADDRESS (City, State, and ZIP Code)

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
IMPLEMENTATION OF A COMPILER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE PHI - Φ (u)

12 PERSONAL AUTHOR(S)
Cole, Eugene J. and Connell, Joseph E. II

13a TYPE OF REPORT
Master's Thesis

13b TIME COVERED
FROM ______ TO ______

14 DATE OF REPORT (Year, Month, Day)
1987 June

15 PAGE COUNT
175

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Functional Language; Applicative Languages; Compiler Design

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis describes the design and implement of a prototype compiler for the functional programming language PHI. The design is highly modularized and the authors think this should facilitate the understanding of both concept and implementation. The front-end of the compiler implements machine independent lexical and syntactic analyzers; top-down parsing techniques are employed. The back-end implements a machine dependent one-pass semantic analyzer and code generator.

Since this implementation is a prototype, it does not possess all of the qualities desirable in a full implementation. The basic contructs of PHI: functions and data definitions are implemented, as well as the integer, natural number, and boolean types. However, the necessary hooks are present and the design is mature enough to allow expanding the prototype to a full implementation.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL
Prof. Daniel Davis

22b TELEPHONE (Include Area Code)
(408) 646-3091

22c OFFICE SYMBOL
Code 52Dv

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

Implementation of a Compiler for the Functional Programming Language PHI — Φ

by

Eugene J. Cole
Major, United States Marine Corps
B. A., The Citadel, 1975

and

Joseph E. Connell II
Captain, United States Marine Corps
B. S., University of Missouri — Rolla, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

ABSTRACT

This thesis describes the design and implement of a **prototype** compiler for the functional programming language PHI. The design is highly modularized and the authors think this should facilitate the understanding of both concept and implementation. The front–end of the compiler implements machine independent lexical and syntactic analyzers; top–down parsing techniques are employed. The back–end implements a machine dependent one–pass semantic analyzer and code generator.

Since this implementation is a **prototype**, it does not possess all of the qualities desirable in a full implementation. The basic constructs of PHI: functions and data definitions are implemented, as well as the integer, natural number, and boolean types. However, the necessary hooks are present and the design is mature enough to allow expanding the prototype to a full implementation.

[illegible]
3769

TABLE OF CONTENTS

I. INTRODUCTION 6
  A. BACKGROUND — GENERAL 6
  B. BACKGROUND — THESIS 7
  C. BACKGROUND — FUNCTIONAL PROGRAMMING 8
    1. Problems with Conventional Languages 9
    2. Functional Languages 10
  D. ASSUMPTIONS 11
  E. CONSTRAINTS 12
II. FRONT–END OF THE COMPILER 13
  A. LEXICAL ANALYSIS — THE SCANNER 13
  B. SYNTACTIC ANALYSIS — THE PARSER 18
  C. ERROR HANDLING 23
III. BACK–END OF THE COMPILER 25
  A. OVERVIEW 25
  B. RUN-TIME ORGANIZATION 25
  C. SEMANTIC CHECKING AND CODE GENERATION 31
  D. OPTIMIZATION 40
IV. RESULTS & CONCLUSIONS 42
  A. RESULTS 42
  B. CONCLUSIONS 43
V. FURTHER RESEARCH 44
LIST OF REFERENCES 46

APPENDIX A THE FUNCTIONAL LANGUAGE PHI — Φ (CONCRETE SYNTAX OF Φ — 10/16/86). 47

APPENDIX B THE FUNCTIONAL LANGUAGE PHI — Φ (CONCRETE SYNTAX OF Φ — 03/03/87) 50

APPENDIX C ASCII REPRESENTATION OF PHI — Φ 53

APPENDIX D THE FUNCTIONAL LANGUAGE PHI — Φ (RIGHT RECURSIVE GRAMMAR) 54

APPENDIX E ROCK COMPILER — HEADER FILES 57

APPENDIX F ROCK COMPILER — MAIN MODULE.. 67

APPENDIX G ROCK COMPILER — SCANNER.. 71

APPENDIX H ROCK COMPILER — PARSER.. 76

APPENDIX I ROCK COMPILER — ERROR HANDLER 106

APPENDIX J ROCK COMPILER — SEMANTIC CHECKER 112

APPENDIX K ROCK COMPILER — CODE GENERATION MODULE........................ 137

APPENDIX L ROCK COMPILER — USER INTERFACE 142

APPENDIX M ROCK COMPILER — RUNTIME UTILITIES.. 145

APPENDIX N TEST SUITE 157

APPENDIX O ROCK COMPILER — USER'S MANUAL 168

INITIAL DISTRIBUTION LIST.. 174

I. INTRODUCTION

A. BACKGROUND — GENERAL

In its attempt to provide students with a well rounded background to the field of computer science, the computer science department at the Naval Postgraduate School offers courses covering recent developments in programming languages. One of the courses deals specifically with the methodology of functional, also known as applicative, programming. Both the theory and the practice of functional programming are covered, concentrating more on the practice than the theory. In order to fully appreciate the nuances of functional programming it would be desirable to provide the students with a functional programming environment. This would provide a first hand look at the fundamental difference in methodologies when programming in functional languages as opposed to programming in traditional imperative languages.

Of the languages currently supported in the department; LISP, on the UNIX[1] environment, comes the closest to meeting this requirement. Although LISP **is** considered a functional language by some, its many extensions and modifications actually brings it into the world of imperative programming. It **is not** a pure functional programming language.

There are several additional problems associated with using LISP to teach techniques of functional programming. Modern LISP dialects do not support all aspects of functional programming. Most notably they lack the ability to define higher–order functions. Dynamic scoping and the semantics of the language make it a pedagogical nightmare to teach.[Ref. 1:p. 0–1] The goal of teaching functional programming would rapidly be overtaken by the necessity of explaining the idiosyncrasies of LISP. In an 11 week quarter, time devoted to LISP would significantly detract from instruction of functional programming.

---

[1] UNIX is a trademark of Bell Laboratories.

Recognizing the shortcomings of LISP, a pure functional language, PHI was developed by Dr. B. J. MacLennan for use in this course of instruction. The syntax of PHI closely follows that of standard mathematical notation. This means students should have little difficulty in learning how to write legitimate PHI statements. Instruction can now concentrate on joining these statements to create functional programs. Hopefully, this will lead to a greater understanding and appreciation of the methodology of functional programming.

B. BACKGROUND — THESIS

Creation of PHI solved the problem of finding a suitable language to use to demonstrate the methodology of functional programming. However, currently PHI programs are programs *on paper* only. There exists no programming environment for the PHI language. So it is impossible to machine execute PHI programs. This thesis attempts to remedy the above problem by providing the first component in a PHI programming environment — a prototype PHI compiler.

Conventional compiler construction techniques were chosen for this implementation for several reasons. By choosing conventional techniques, the authors were able to address the problems associated with utilizing conventional methods for implementing a compiler for a functional language[2]. Additionally, realizing that both the language and system would change, the authors wanted a well documented and understood methodology. The cost of maintaining a system can be as much as three times the development cost [Ref. 2:p. 478]. Therefore, it was imperative to choose a methodology that supported a clean and structured design.

---

[2] Specific problems and solutions are covered later in Chapters Two and Three

Following conventional methodologies, the authors separated the PHI compiler design into a front–end[3] and a back–end[4]. The overall general design of the PHI compiler is shown in Figure 1.1. The front–end, containing the scanner (lexical analyzer) and parser (syntactic analyzer) is essentially responsible for analysis of the external file containing the source program. The PHI compiler back–end couples semantic analysis with code generation to produce code suitable for execution on the target machine. [Ref. 3:pp. 5–6] The authors felt that a clear and distinct separation between parts would aid understanding of the system, simplify division of labor, and increase ease of development and maintenance. It should also result in greater flexibility for follow–on development in the PHI programming environment. As an example, the current front–end could be modified to support a PHI interpreter.

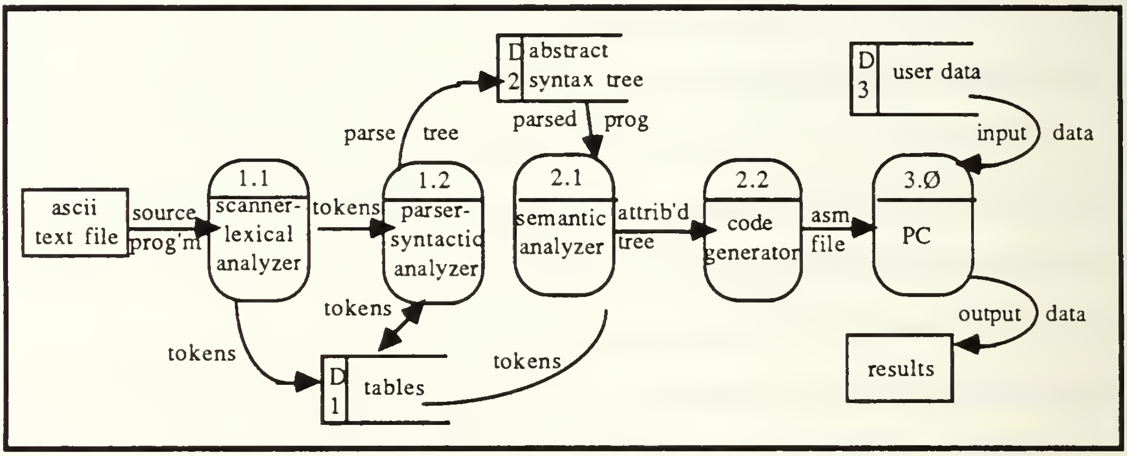

D 2 abstract syntax tree
D 3 user data
parse tree
parsed prog
input data
ascii text file
source prog'm
1.1 scanner-lexical analyzer
tokens
1.2 parser-syntactic analyzer
2.1 semantic analyzer
attrib'd tree
2.2 code generator
asm file
3.0 PC
tokens
tokens
tokens
output data
D 1 tables
results

Figure 1.1 General Design of the PHI Compiler

C. BACKGROUND — FUNCTIONAL PROGRAMMING

Functional programming is a methodology in favor among academicians. Although applicative programming goes further back, it is generally agreed that, as a methodology, functional programming traces its roots to John Backus [Ref. 4:p. 404, Ref. 5:p. 65]. In his acceptance speech for the 1977 ACM Turing Award, Backus criticized traditional programming languages and programming styles. He went on to propose a new methodology of programming that involved "the use of a fixed set of combining forms called functional forms." [Ref. 6:p. 619] This methodology is known today as functional programming.

---

[3] Design and implementation of the front–end is discussed in Chapter Two.

[4] Design and implementation of the back–end is discussed in Chapter Three.

1. Problems with Conventional Languages

Backus feels [Ref. 6:pp. 613–619] that the basic underlying problem with conventional languages is the existence of the assignment statement. The assignment statement plays a central role in conventional languages and breaks programming into two worlds. Backus calls the right–hand side of assignment statements, expressions, the first of these worlds. The second world is the world of statements, with the primary statement, of course, being the assignment statement.

Several problems are associated with assignment statements. First, they permit programs to be held hostage through access to their variables. Since variables are used to imitate the machine's storage cells; assignment statements allow, even encourage, state changes to take place. This access, either direct or indirect, permits such problems as side effects, unintentional state changes, and aliasing to arise. It then becomes difficult to reason about the correctness of these programs, so proving simple programs correct is an arduous task and proving complex programs correct is virtually impossible. Additionally, by permitting the value of variables to be changed, the assignment statement makes temporal order of execution of statements critical. For example, the following two pieces of code produce dramatically different results depending on which statement inside the for loop is executed first.

```
for (i = 0; i != some_value; ++i)
{       if( i % 2 == 0)
                continue;
        DoSomething(i);
}
```

```
for (i = 0; i != some_value; ++i)
{       DoSomething(i);
        if( i % 2 == 0);
                continue;
}
```

These problems interact so that it becomes extremely difficult to create new programs out of old ones. [Ref. 6:pp. 613 - 619, Ref. 1:pp. 1–2 - 1–20]

Another problem associated with assignment statements is that each produces only a one–word result. In effect, they force programmers to think in a word–at–a–time manner. For example, to apply a function to an entire array of values, the programmer must access each value individually. Not only is this wasteful of computer assets, but it results in what Backus refers to as the "von Neumann bottleneck" of conventional programming languages. [Ref. 6:pp. 613 - 619]

2. Functional Languages

Backus proposes the methodology of functional programming as the solution to these problems. Functional languages have removed variables and the assignment statement from their syntax so that their basic building block becomes the function. It is through "the use of a fixed set of combining forms...plus simple definitions" [Ref. 6:p. 619] that the programmer is able to build new functions from existing functions. It thus becomes possible to form a new program by combining two or more existing programs or functions together.

The absence of assignment statements and variables removes the problems plaguing conventional languages caused by side effects, etc. because the program now operates exclusively in the world of expressions. This permits the programmer to maintain a clear conceptual view of the program. It is easier to understand and reason about the task the program is to perform [Ref. 5:pp. 65 - 69]. It now becomes not only possible, but practical to prove programs correct [Ref.6:pp. 624 - 625].

Another direct benefit stemming from the absence of side effects is order. The values of expressions are no longer dependent on the order in which they are evaluated. Therefore, functional languages provide a natural means of performing parallel computations [Ref. 7:p. 35]. Functional languages and the associated methodology of functional programming may very well provide the key to programming the massively parallel computers entering service nowadays. All of the above benefits have applicability to ongoing research in the SDI program.

The authors feel that functional programming can best be summarized by the following thought — assignment statements are to functional programming what GOTO statements are to structured programming.

D. ASSUMPTIONS

An IBM[5] personal computer/IBM compatible personal computer was chosen as the target machine for this implementation. The authors felt that the nature of the language and its intended use were better suited for the PC/personal work station environment as opposed to a mini– or main–frame time shared environment. The PC environment should provide greater flexibility and freedom when implementing follow–on tools for the PHI programming language. Also, future compiler improvements will not have to be concerned with extraneous interfaces to another system. Working with a PC environment eliminates the need to take into account the effects the PHI environment will have on another user of the system. The implementor is able to work with a system that remains constant — a known quantity.

The assumed target machine configuration is based on the equipment available in the Naval Postgraduate School's computer science microcomputer lab. Each machine is configured with 640K bytes of RAM, one (most have two) 20M byte hard disk drive, one 1.2M byte 5 inch floppy disk drive, and the 8087 math co–processor; each currently operates under the MS–DOS[6] 3.x operating system. These machines are readily available to all computer science students at the Naval Postgraduate School, and many students own personal computers with similar configurations. It is not necessary to utilize a hard disk when executing the PHI compiler.

---

[5] IBM is a registered trademark of Internal Business Machines Corporation.

[6] MS–DOS is a registered trademark of Microsoft Corporation.

E. CONSTRAINTS

As is the case with most implementation theses, time was probably the biggest constraint facing the authors. This involved making certain trade–offs; e.g. should the major effort be directed towards a full implementation of PHI while concentrating on a particular component of the compiler, or should the major effort be directed towards a full implementation of the compiler while concentrating on a subset of the PHI language? The authors felt that the greatest benefit could be gained by implementing a complete compiler. Having to actually face the issues and problems associated with designing, implementing, and interfacing a full compiler implementation would be much different than just reading about them in a text. As a result, this thesis implements only a subset[7] of PHI.

Since PHI is an experimental language it is still undergoing changes and revisions. Trying to modify and update the compiler design with each version proved to be an impossibility. The authors were forced to freeze the design based on the language as it stood on 07 January 1987. Any follow–on work will need to update the front–end and back–end of the compiler to meet the requirements of these new versions of PHI. A description of the grammar as implemented and a description of the latest version of the grammar may be found in the Appendixes.

---

[7] This subset is discussed in the individual chapters on the front–end and back–end.

II. FRONT-END OF THE COMPILER

The authors separated the design of the PHI compiler into two modules, a front–end and a back–end. These modules were then further subdivided to produce the general layout of Figure 1.1. The authors believe this modularization simplifies the design and will aid in understanding the system, thus decreasing future maintenance problems.

The front–end of the PHI compiler is comprised of the scanner (lexical analyzer), the parser (syntactic analyzer), and their associated error recovery routines. Two possible interactions between the lexical and syntactic analyzers were considered. The first incorporates the scanner into the parser, and tokens are produced by the scanner only upon request of the syntactic analyzer. Thus, this system acts like a pipeline. An alternate method is to allow the scanner to tokenize the entire source program, store the tokens in some data structure, and pass this structure to the parser. [Ref. 3:p. 10]

For the prototype implementation of a PHI compiler, the authors based the design on the first interaction. Although the second method is conceptually very easy to understand, the authors think the current implementation is clean and will readily lend itself to future enhancements. Any input alphabet peculiarities are restricted to the lexical analyzer, and this independence should provide benefits for the next student(s) who work on the PHI programming environment.

A. LEXICAL ANALYSIS — THE SCANNER

The PHI compiler reads a source file of ASCII text which is fed to the scanner for lexical analysis. The principle task of lexical analysis is to separate or divide the source program into tokens for use during syntactic analysis [Ref.8:p. 84, Ref. 9:p. 155]. This is accomplished in the PHI compiler through a character–by–character examination of the user's source file. These characters are assembled/grouped into the individual tokens which represent terminal symbols of the PHI grammar. Examples of some of the terminal symbols are operators, identifiers, keywords, and constants. A complete listing of the PHI tokens may be found in the header file for the scanner in Appendix E.

The primary advantage to tokenizing the source program is that the design of the syntactic analyzer needs to take into account only one type of data unit — the token [Ref. 3:p. 7]. This simplifies the design of the parser because provisions do not have to be made for handling white space and comments. The scanner has already removed them. Also, removing white space and comments and utilizing a fixed–length representation for the tokens saves space. Once tokenization is complete, the source program can be discarded and the compacted tokenized file can be utilized for further analysis.

In order to correctly tokenize the source file there must be some discrete means available to accurately represent each token. There are several ways of describing tokens. One means available is to use a regular grammar. In this method "generative rules are given for producing the desired tokens" [Ref. 3:p. 142]. An equivalent, but different, method is to use finite–state acceptors, FSAs, to recognize tokens. The authors found it easier to visualize this as a recognitive vice generative problem. For this reason the various tokens were modeled using FSAs. An example of an unsigned number recognizer is shown in Figure 2.1. The interested reader is directed to Tremblay and Sorenson [Ref. 3:Chapter 4] for an excellent introduction to the practice of using FSAs to model tokens. The authors found that utilizing FSAs greatly simplified the design, coding, and debugging of the lexical analyzer — one picture was worth a hundred lines of code.

The ideal grammar would allow each token to be uniquely and unambiguously identified. Once the lexical analyzer started on the path of building a token, it would be able to continue until the end with no backtracking. Due to limitations with the standard

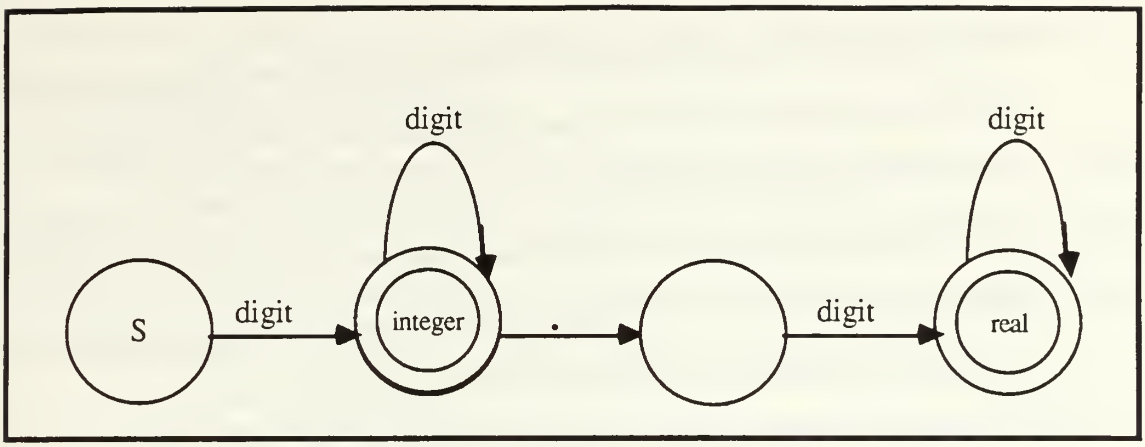


Figure 2.1 Unsigned Number Recognizer

ASCII character set, the designer of PHI used multiple keystrokes, or characters, to represent various operators in the language[8]. This resulted in compound token types. Also, as in other programming languages, PHI overloads certain operators, allowing them to do double duty[9] by taking on different context–dependent meanings.

The problem of dealing with compound token types was easily handled through the use of a single lookahead character. For example, upon finding the character "-", the scanner looks ahead to the next character to see if it is ">" (→) or another "-" (--). If the next character is neither of these two, it indicates that the token is just the simple token "-". Distinguishing overloaded operators was solved by essentially ignoring it in the scanner! The authors took the position this is basically a syntax analyzer problem and there was no reason to complicate the scanner by handling it. The scanner just identifies a generic token type, e.g. SUB_, and lets the parser make the proper determination of its true meaning,e.g. SUB_ or NEG_.

There are several design decisions relating to the lexical analyzer worth noting. The authors, following the example of Pascal, C, and other languages, took the position that

---

[8] Some examples of this are -> for →, == for ≡ and <> for ≠.

[9] For example, + and - can serve as either an unary or binary arithmetic operator.

PHI's keywords[10] are reserved words and may not be redefined and used as identifiers. Alternate decisions would have been to distinguish keywords from identifiers based on context, as PL/I does, or to precede them by some special character, as ALGOL 60 and ALGOL 68 do [Ref. 3:p. 91]. PHI has a very small set of keywords, smaller than C's, and the authors think that this decision makes life easier for the programmer by simplifying debugging of programs. It certainly made life easier for the authors.

PHI's grammar makes no provisions for programmer comments. The authors originally implemented comments by requiring the programmer to explicitly indicate the beginning and end of each comment with a special character. After scanning the special character at the beginning of the comment, the lexical analyzer would ignore all following characters until the special character was once again found. Following conversations with PHI's designer this implementation was changed. Comments are now implemented the same way they are in Ada[11]: the comment terminator is the end–of–line character. Not only did this simplify the recognizer for comments, but it also completely removed the problem of runaway comments.

A name table is used to point to the names of all identifiers and constants. A symbol table was originally utilized but later discarded when the authors realized the syntax of PHI makes analyzing an abstract syntax tree easier than analyzing a flattened tree. The information normally associated with a symbol table is now held in the nodes of the tree. This permits just the first instance of each name to be placed into the name table. In other words, regardless of how many times and in how many scopes the identifier X is used, X appears only once in the name table. The token returned to the parser would indicate a token type of identifier and the parser would then know to dereference the pointer to find the string containing the actual name, X.

---

[10] A complete listing of PHI keywords may be found in the header file for the scanner in Appendix E.

[11] Ada is a trademark of the Ada Joint Programming Office, Department of Defense, United States Government.

Because keywords are reserved, each potential identifier must first be compared against the possible keywords prior to being placed in the name table. The authors implemented a keyword table to simplify this process. Knuth [Ref. 10:pp. 406-410] has shown that a binary search is the most efficient way of searching an ordered table, using only comparisons. For this reason the keyword table is kept in alphabetical order. The lookup, which is at worst O(log n), is performed using a binary search of the keyword table.

In an attempt to improve the efficiency of the name table, the authors implemented it as a hash table. McKeeman [Ref. 11:pp. 253–301] experimented with six different length dependent hash functions. He found that the function producing the best results involved summing the internal representation of the first and last characters of the variable's name with its length shifted four places to the left. This was the function utilized by the authors. The possibility of collisions is reduced by choosing a prime number as the table size. However, since this only reduces, not eliminates, the possibility of two or more names hashing to the same value; the authors had to make provisions for handling collisions.

A variant of the chaining method of collision–resolution was chosen. In PHI's implementation, each of the name table slots/buckets holds a data structure that can contain both the name of the variable and a pointer to another structure of the same type. So each hashed value points to a linked list of names. This method offers the advantage of providing better performance than linear probing [Ref. 12:p. 89], is conceptually easy to visualize/work with, and also solves the problem of possibly overflowing the hash table. It does require slightly more memory to implement, but the authors determined that the benefits of this method far outweighed the slight increase in storage requirements. [Ref. 12:pp. 83–93]

B. SYNTACTIC ANALYSIS — THE PARSER

The purpose of the parser is twofold: 1) to determine if the program, as represented by the output from the scanner, is syntactically correct; 2) to impose a hierarchical structure on the token stream, fitting it into the abstract syntax tree which is the output of the parser [Ref. 8:pp. 7–8, Ref. 9:p. 7]. Traditionally, these tasks are done by either a top–down or bottom–up methodology [Ref. 8:p. 41]. Both methodologies use the tokens generated through lexical analysis.

The terminology top–down refers to the order in which the nodes of the parse tree are constructed. Top–down parsing starts from the root of the tree and proceeds downward towards the terminal symbols at the leaves. The parse tree is constructed from the top to the bottom by applying *productions* of the grammar to generate strings of terminals and nonterminals. On the other hand, bottom–up methodologies start from the terminal symbols at the leaves and proceed upwards to the root. The parse tree is constructed from the bottom to the top by applying *reductions* of the grammar to generate single nonterminals from strings of terminals and nonterminals. [Ref. 8:pp. 40–41, Ref. 9:pp. 134–136]

It is generally agreed that the popularity of top–down parsing techniques is "due to the fact that efficient parsers can be constructed more easily by hand". [Ref. 8:p. 41] The authors can attest to the fact that the concept of top–down parsing is very easy to grasp. When parsing PHI, it is natural to begin with the start symbol of the grammar, BLOCKBODY, and work forward from there to analyze the token stream. So, partially because of its efficiency, but primarily because of its ease of understanding and use, the authors chose the top–down methodology of recursive–descent parsing to design and implement the syntactic analyzer.

In recursive–descent parsers, separate procedures/functions are written to recognize each nonterminal of the grammar [Ref. 3:pp. 219–220]. This technique gets its distinctive name "because nonterminals can appear in the right–hand sides of each other's productions, the procedures for recognizing nonterminals are recursive." [Ref.9:p. 150] To state it more clearly, the function to recognize nonterminal 'A' could end up calling itself recursively if either 1) 'A' appears on the right–hand side of the production describing 'A' itself, or 2) 'A' appears on the right–hand side of the production describing another nonterminal 'B' **and** 'B' appears on the right–hand side of the production describing 'A'. Regardless of how one looks at the nature of the technique, one usually identifies a stack with recursion. What made this technique so easy to implement was that the authors were able to use C's underlaying mechanism for handling recursive functions. The authors did not have to *explicitly* maintain a stack of symbols for each function call; instead, the information was *implicit* in the stack of activation records resulting from each function call.

Top–down parsing techniques, especially recursive descent, offer straightforward means of implementing a syntactic analyzer. However, these techniques are applicable only to a subset of the context–free grammars and it is **essential** that all left recursion be eliminated from the grammar [Ref. 3:p. 211]. In other words, there must not exist any productions describing nonterminal 'A' with 'A' appearing as the first element on the right–hand side of the production. Obviously, if this situation existed, it would be possible to present the parser with strings to parse that would cause it to enter "an infinite loop of production applications" [Ref. 3:p. 211], never to be heard from again. The PHI production QUALEXP = QUALEXP **WHERE** AUXDEFS is an example of this type of string. The parser would hang up looking for QUALEXP and would never leave this loop until the machine ran out of memory stacking activation records. In order to employ top–down parsing techniques with PHI the authors rewrote the PHI grammar to be right–recursive[12]. However, as shown below, even the new grammar does not lend itself to "pure" recursive descent parsing techniques.

---

[12] The right recursive syntax of PHI may be found in Appendix D

From the compiler writer's point of view the ideal grammar would allow the correct production rule to be applied in every step of the parsing process. Constructing the parse tree would then proceed in a completely deterministic manner. When this is not possible, there are two basic parser design methods for dealing with nondeterminism in the grammar [Ref. 9:pp. 151–152]. In the backtracking method, which is generally not applicable to recursive–descent techniques, the parser picks an arbitrary production and continues with the parse [Ref. 9:p. 151]. If the parse is successful it is assumed that the correct production was chosen. However, if an error is later discovered, the parser *backtracks* to the last choice, a new production is chosen, and the parser presses forward again. This process continues until either the parse is successful or the parser runs out of possible productions to chose from. The second method requires a modification to the grammar which results in a deterministic parser: the grammar is rewritten using a process called left factoring to avoid choices among nonterminals [Ref. 9:p. 151].

For the most part, the design of PHI is conducive to recursive descent parsing techniques. There are, however, several productions where this is not so. The result was that a degree of nondeterminism arose in the parser design. The authors attempted to solve this problem through a combination of left factoring and the employment of a simple single token look–ahead. This solution worked for all but the two productions described below. In one case a two token look–ahead was employed and backtracking was used tn the other. This is not to say that the authors are absolutely certain that PHI is **not** an LL(1) grammar or that backtracking **had** to be used. These solutions were used because they solved the problem at hand.

A two token look–ahead was used for the production[13] ARGBINDING = [QUALEXP OP]. When the token '[' is found, a flag is set to indicate that an ARGBINDING is being parsed. The first look–ahead token is utilized when parsing the QUALEXP part. QUALEXP, for example, may parse as TERM, which in turn may parse as either FACTOR or FACTOR*TERM. After succeeding on FACTOR, a look–ahead is employed to look for the MULOP, *, to see if a recursive search for another TERM should be initiated. This methodology works as long as QUALEXP was not called from ARGBINDING. If it was called from ARGBINDING, argbinding flag set, the operator * could be the trailing operator in the ARGBINDING production and not part of the TERM production. In order to make this determination, an additional look–ahead is utilized to look for the token ']'. If ']' is found the QUALEXP production is terminated, e.g., term does not recursively call itself again, and the ARGBINDING production is allowed to proceed to completion.

---

[13] A complete description of the PHI grammar may be found in the Appendices

Backtracking was utilized when parsing productions of ACTUAL: ACTUAL = COMPOUND and ACTUAL = DENOTATION = FORMALS |-> ACTUAL. Legitimate PHI sentential forms produced by the production FORMALS = (FORMALS^+) are proper subsets of the sentential forms produced by the production COMPOUND = (ELEMENTS), excluding the empty compound statement. Since any number of identifiers may appear between the parentheses, it is not practical during the parse to utilize look–ahead to determine the presence of the token "|->". In effect, the parser first realizes it was parsing a DENOTATION when it finds "|->". This problem was solved by designing the parser to apply first the compound production when presented with this choice. If "|->" is later found, the parser then backtracks[14] to the FORMALS production. The normal costs associated with backtracking were not evident in this isolated case. As described below, space trade–offs had previously been made and the parser was already working with an abstract syntax tree. The root to the subtree containing the previously parsed compound was simply passed to the FORMALS production to insure that the string could have been produced by FORMALS. After ascertaining FORMALS, the parser now continues the parse using the DENOTATION production.

---

[14] A purist would say that this instance of backtracking means that the PHI compiler does not in fact employ a recursive–descent parser.

The production QUALEXP = QUALEXP **WHERE** AUXDEFS required a deviation from pure recursive descent parsing. The semantics of this production are such that a terminal (e.g., an identifier) may be used prior to its definition. In itself, this does not present a major problem for the compiler writer. However, this construct also changes the scope of the identifier since the *inner–most* scope, in the form of the QUALEXP, is parsed first and the parser then works its way to the *outer–most* scopes, the AUXDEFS. This problem is analogous to that of mutual recursion in Pascal, without the benefit of the forward declaration [Ref. 4:p. 213].

Originally, the parser was designed to output the parse tree in flattened form, essentially a post–order walk of the tree. This design implemented traditional symbol–table management routines. However, after obtaining a clearer understanding of the semantics involved with the problems mentioned earlier, notably the production QUALEXP = QUALEXP **WHERE** AUXDEFS, the authors realized a traditional symbol–table would be too inefficient. Management of the table would take an inordinate amount of assets and be too unwieldy to work with. The authors solved this problem by maintaining the status of the parse in an abstract syntax tree so the output from the parser is now in tree form. This permits information originally held in the symbol–table to be maintained in the tree itself. The parser is able to analyze the source program by *walking* the tree and *decorating* the nodes with required information. Maintaining a binary tree in memory does require more space, but this is insignificant when compared with the benefits.

Interestingly, maintaining the parse in tree form presented several additional benefits. The solution to the aforementioned problem of distinguishing between COMPOUND and DENOTATION became trivial because it was now simply a matter of returning to the appropriate subroot and rewalking the tree. Also, working with a binary tree permitted the authors to perform a modicum of optimization in the parser. It becomes relatively straightforward to perform compaction on an actual tree.

The authors think that this design offers maximum potential for future enhancements of the PHI programming environment. One possibility would be to use this front–end to drive a PHI interpreter. Modularization of the front–end in this manner simplifies functional understanding of the front–end and should lead to increased ease of maintenance and portability. To demonstrate portability, the authors recompiled the front–end and executed it on a 68000 based processor. This was accomplished with no modifications to the source program, just replacement of C run–time header files for the new target machine.

C. ERROR HANDLING

Tremblay and Sorenson [Ref. 3:p. 183] classify error responses into three categories:

I. Unacceptable responses
  1. Incorrect responses (error not reported)
     a. Compiler crashes
     b. Compiler loops indefinitely
     c. Compiler continues, producing incorrect object program
  2. Correct (but nearly useless)
     a. Compiler reports first error and then halts

II. Acceptable responses
  1. Possible responses
     a. Compiler reports error and *recovers*, continuing to find later errors if they exist
     b. Compiler reports the error and *repairs* it, continuing the translation and producing a valid object program
  2. Impossible with current techniques
     a. Compiler *corrects* error and produces an object program which is the translation of what the programmer intended to write

In the prototype PHI compiler, the authors have implemented a limited form of error *recovery*. The primary benefit of error recovery is to "prolong the compilation life of the program as long as possible before the compiler gives up on the source program". [Ref. 3:p. 11] This allows the maximum number errors to be discovered per compilation, shortening the edit, compile, debug cycle inherent to writing computer programs.

The authors analyzed the intended environment and use of the PHI compiler and decided that lexical analysis and syntactic analysis were the most likely source of errors.

Lexical errors basically involve invalid characters or incorrect tokens. Common examples of these types of errors are unrecognized words, misspelled identifiers/keywords, or illegal characters. Syntactic errors relate to incorrect structure of the program. These errors arise when the programmer failed to follow the rules, productions, of the grammar. The form of the program is wrong. [Ref. 9:p. 226, Ref. 3:p. 185]

One thing the error handler should **not** do is exacerbate the situation by reporting bogus errors or executing an erroneous program. To insure erroneous programs are not executed, the authors inhibited object file production if any errors were discovered. The authors do not believe the compiler should allow code generation to continue, or even begin, if the source program has errors. Often times one error leads to an avalanche of errors being reported and this is extremely annoying to the programmer. The authors attempted to minimize this situation, but found it impossible to eliminate completely because some errors feed on others. To insure the programmer would not become overwhelmed with error messages, the authors terminate the compilation after 10 errors. Also, for programmer convenience, actual error messages are outputted instead of error codes. The authors saw no justification in using a cryptic code when a plain language message served much better. Since the authors anticipate students in functional programming classes to be primary users of the PHI compiler, error messages have their basis in the productions describing the PHI language. It is assumed that users of the PHI compiler have an understanding of PHI's syntax.

III. BACK–END OF THE COMPILER

A. OVERVIEW

The back–end of the compiler consists of the semantic checker and code generator. Semantic checking and code generation are completed in one pass, and the output is a sequence of bytes, held in memory, which correspond to ASCII characters. These characters are then written to a text file, which the assembler uses to output an object file. This output is linked to the appropriate run–time routines to make a usable program. For the current implementation, a RASM86 assembler and LINK86[15] linker are used.

B. RUN-TIME ORGANIZATION

Since PHI is a structured language with scoping and function calls, it lends itself to a stack–oriented run–time architecture. The stack is set up to accomplish two tasks: 1) to hold pointers to the current operands, and 2) to hold activation records for functions currently in use. Both of these tasks are described below.

There is a 64 kilobytes limit on memory used while a program is running. This limitation is imposed because the memory is addressed as an offset from a base address, and the maximum offset is 64K. This space is competed for by the stack, current variables, and constants (see Figure 3.1). The stack grows from the top of this space down, and the variable space grows from the base of this space up, preventing wastage by either component. Because PHI is a functional language, a value is returned from each operation, and a pointer to this value is placed at the top of the stack. The returned value is placed in the lowest available space in the part of memory assigned to variables and constants. A heap allocation method is not currently used because 1) all data types currently implemented use only one word of memory, and 2) there is no fragmentation of memory because all types are currently static. If the next operation is a binary operation, a pointer to the second operand is placed on the stack, and the operation takes place using the two topmost pointers. The result is placed in memory, and the process begins afresh with new operands. If the next operation is unary (such as the negation operation), no change to the stack takes place and the variable in memory is altered as the program directs.

---

[15] RASM86 and LINK86 are trademarks of Digital Research, Inc.

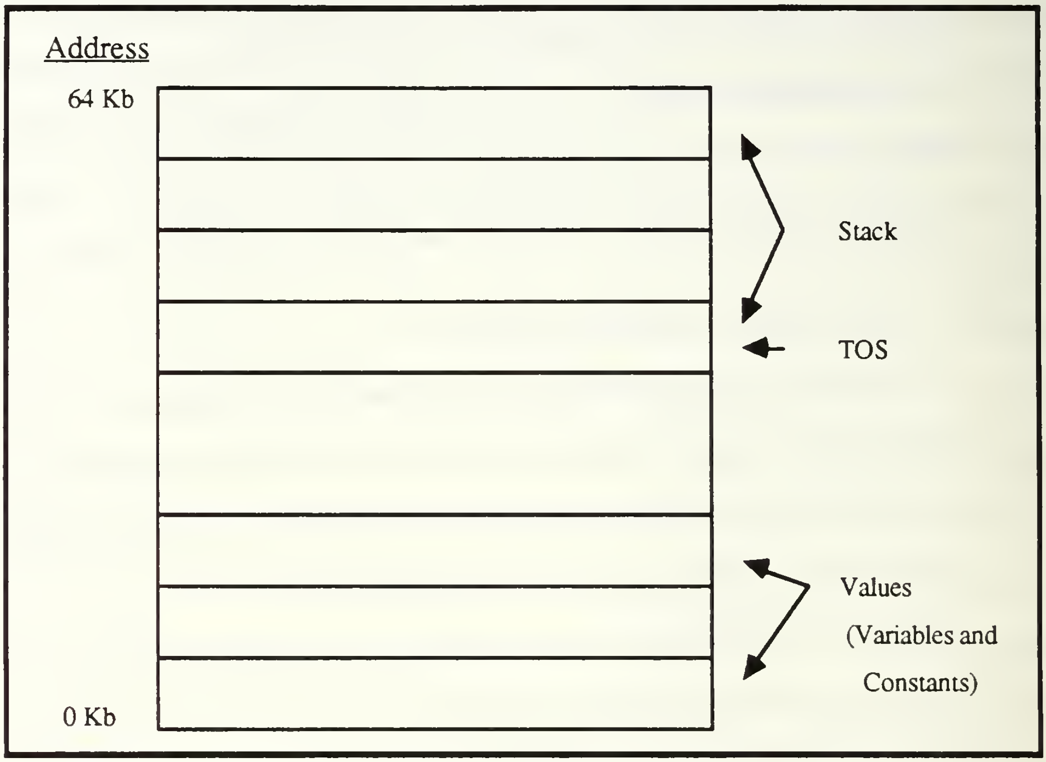


Figure 3.1
Memory Organization

If the second operand of an operation is to be the result of a function call (e.g., "2 * f(x)"), an activation record is placed on top of the pointer to the first operand and the function's value is calculated. Then, the activation record is deleted and a pointer to the function result is saved and placed at the top of the stack.

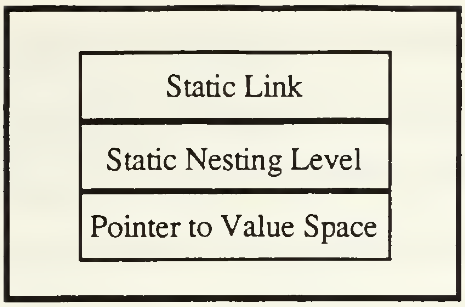
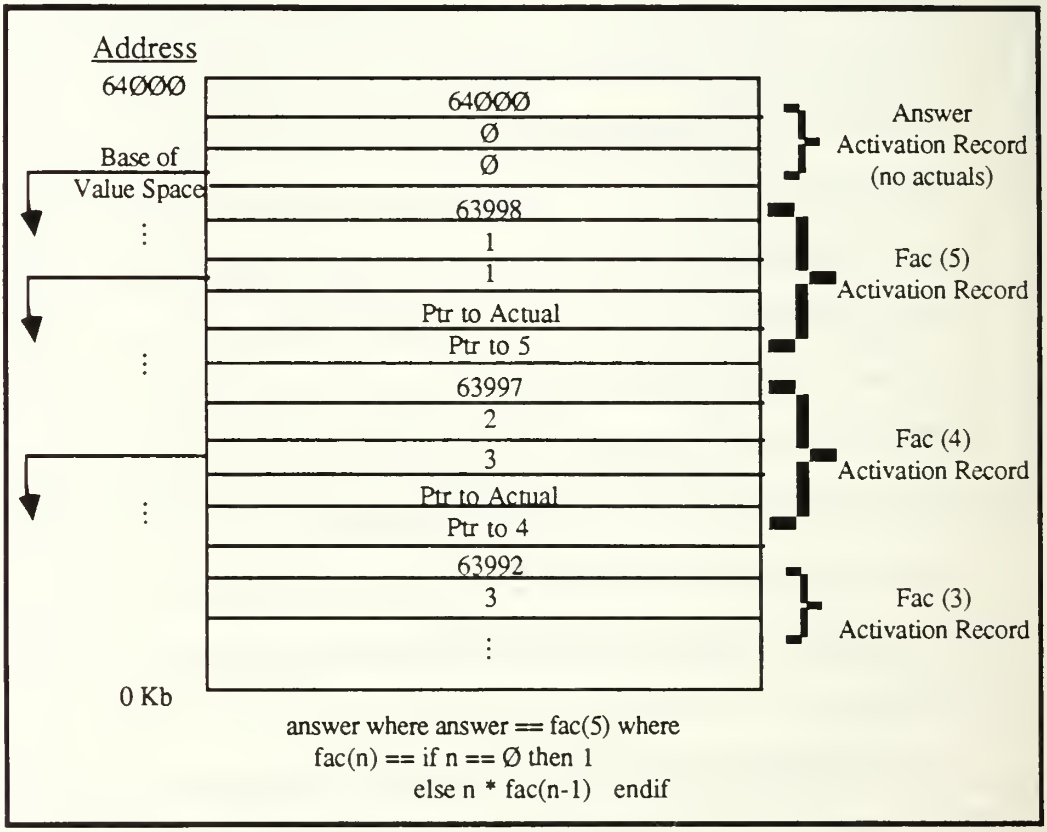


Figure 3.2
Activation Record

The activation record itself, Figure 3.2, contains three parts: the static link, the static nesting level, and a pointer to the address in memory where the function's first variable is stored. The static link is a one–word pointer which points to the static nesting level space of the previous activation record, and is used to traverse the stack from activation record to activation record, i.e. a static chain. [Ref. 4:p. 77]. The static nesting level and the pointer to the base of the storage space for a scope's values are used to access variables and constants. In this design, a two-tuple (**B, L**) is associated with each variable. In this two-tuple, **B** represents the static nesting level and **L** is the offset within that level. By following the static chain for (current nesting level - target nesting level) links, the activation record of the scope of the target value can be accessed. Then, the address of the variable is calculated by adding **L** to the low address of the scope's variables. An alternate method would have been to store the values directly in the stack between or within activation records. However, this is a messy process when dealing with dynamic data structures such as sequences. Additionally, it is conceptually easier to divide the stack and the variables.

Functions are implemented as calls to assembly language subroutines, with pointers to the arguments placed on the stack before calling the routine. Using this scheme, and noting the fact that PHI cannot have side effects, the implementation of recursion is straightforward. Whenever a function is called, its activation record is placed on the stack and pointers to its arguments are placed on top of the activation record. If the function is recursive, the assembly language subroutine simply calls itself until the base of its recursion is reached or until stack overflow is reached. Figure 3.3 shows an example of a series of activation records called by a program with a recursive function. Note that the data definition ("answer") has no arguments and simply calls the factorial function. The factorial function, on the other hand, has an argument and it uses that argument as an operand. So, a pointer to that value is put on the stack and the next operand, fac (n - 1), is put on the stack as an activation record. When fac (n - 1) is evaluated, a pointer to its return value is placed on the stack. This cycle of evaluation, pop activation record, evaluation will continue until the data definition "answer" is evaluated.


Address
64000
Base of
Value Space
0 Kb
64000
0
0
63998
1
1
Ptr to Actual
Ptr to 5
63997
2
3
Ptr to Actual
Ptr to 4
63992
3
Answer
Activation Record
(no actuals)
Fac (5)
Activation Record
Fac (4)
Activation Record
Fac (3)
Activation Record

```
answer where answer == fac(5) where
   fac(n) == if n == 0 then 1
             else n * fac(n-1)   endif
```

Figure 3.3
Factorial Program and Activation Records

As an example of the code generated for function calls and recursion, the following PHI program fragment is used : C (n) == if n = 0 then 1 else C (n - 1) * n endif.

This, of course, simply calculates the factorial of the integer n. Figure 3.4 is the listing of the assembly language segment which is generated from this fragment.

```
Address/Machine Code        Assembly Language

   0103 E94A00                    0150  jmp a10000
                                        a10001:
   0106 B90000                          mov cx,0
   0109 E80000              E           call i_formal
   010C B80000                          mov ax,0
   010F E80000              E           call iputvalue
   0112 E80000              E           call iequ
   0115 E80000              E           call igetvalue
   0118 3D0100                          cmp ax,1
   011B 7509                      0126  jne a10003
   011D B80100                          mov ax,1
   0120 E80000              E           call iputvalue
   0123 E92600                    014C  jmp a10002
                                        a10003:
   0126 B90000                          mov cx,0
   0129 E80000              E           call i_formal
   012C B90000                          mov cx,0
   012F E80000              E           call i_formal
   0132 B80100                          mov ax,1
   0135 E80000              E           call iputvalue
   0138 E80000              E           call isub
   013B E80000              E           call ppop
   013E 51                              push cx
   013F 57                              push di
   0140 BB0100                          mov bx, 1
   0143 E80000              E           call i_mov
   0146 E8BDFF                    0106  call a10001
   0149 E80000              E           call imult
                                        a10002:
   014C E80000              E           call del_scope
   014F C3                              ret
                                        a10000:
```

Figure 3.4
Assembly Language Output from Factoral Program

The label "a10001" at address 0103 is the label of the subroutine which returns the factorial. When it is called, pointers to the values of the arguments are placed on the stack. If the subroutine is called before the base of the recursion is reached, a jump is made to label a10003. Then, the new actual value (n - 1) is calculated and placed in the low part of memory, a pointer to the value is put on the stack, and the values are prepared for calling by the next subroutine (lines 0126 to 0143). The factorial subroutine is then called again. This process continues until the base of the recursion is reached; in this case a pointer to the integer value is put at the top of the stack (line 011D), and a jump is made to label a10002. Here, the subroutine "del_scope" tears down the activation record on the stack and puts a pointer to the result of the function at the top of the stack. Clearly, recursion in the PHI program can be implemented by a parallel recursion in the assembly language output of the compiler.

Another feature of the output code shown in Figure 3.4 is that there is an unconditional jump around the function (lines 0103 and 014F). This is a result of the decision to output inline code in spite of the fact that functions can be called at random. There are both space and time penalties to be paid for these jumps, especially since each function must have a jump and label instruction bracketing it. However, the ultimate effect of all these jumps is to get to the label at the bottom of the program. The result is that all but one jump/label pair could be eliminated by an optimizer, making the penalty trivial. Another solution considered was to generate code for functions and the "main" program separately, then combine the two when printing the output from the code generator. This was not done for reasons put forth in the section that describes the semantic analyzer.

Variable and constant storage is word oriented rather than byte oriented to take advantage of the 8086 processor's 16 bit capability. Integers and naturals are both represented as single words, and booleans are represented as integers, either 1 or 0. While this boolean representation is somewhat wasteful in terms of memory space, it allows for a great deal of overlapping in certain subroutines used in function calling and comparisons. It is planned to represent real numbers with two words of memory, and sequences using linked lists. Neither of these types have been fully implemented; however, there are provisions in the compiler for adding these features at a later date.

There is currently no dynamic allocation of registers. Some registers are used for specific purposes; for instance, the SI register is used to mark the top of the program stack, and of course the BP and SP registers are used to manage the machine's stack. In general, arithmetic processes take place in the AX register, using other general registers as auxiliaries as needed. When variable space is needed, the highest unused address space is allocated and, when a function is finished, only the result is saved in storage; all other value spaces are returned for use by the program.

Error handling is probably the simplest part of the run-time routines. Any run time error such as overflow or division by zero errors will result in an appropriate error message to the user (see Appendix O for a full listing of error messages). Then, program execution will terminate and control is returned to the operating system.

C. SEMANTIC CHECKING and CODE GENERATION

The PHI compiler utilizes the recursive descent technique to perform semantic checking and code generation in one traversal of the parser tree. In most cases, tree nodes are filtered through the **semcheck** function, which calls various procedures based on the name of the node. These procedures, in turn, call **semcheck** for each of their children, and the process is repeated until the leaves of the tree are reached. The function **semcheck** then returns a type (e.g., integer, real, boolean), which the parent node uses to determine the semantic correctness of its subtree. With the information returned from the **semcheck** function, the parent procedure can do one of three things: return a type, convert one node to a different type, or declare an error condition.

Concurrent with semantic checking, code is generated. As noted above, this is assembly language code written to a buffer in memory. If an error condition is declared, however, a flag is set and code generation ends. Semantic checking will then continue until the tree is completely traversed or ten errors are accumulated; then, the semantic checking process terminates. Unlike the parser, the semantic checker makes no attempt at error recovery; top–down checking simply continues normally from where the error was detected.

Top–down semantic checking results in a neat, trim package for the back end of the compiler. Unfortunately, there are some problems that pure top–down checking will not solve. For instance, determining if there is a one–to–one match between formals and actuals for a given function involves some detours from top–down checking, as explained below.

The scoping rules of PHI provided the largest challenge to writing the semantic checker. One solution is a multiplicity of stacks. The size of these stacks depends upon the number of its constituents visible at any one time. Usually, the proper match for an item is the one found closest to the top of the stack. However, because of the semantics of the "and" construct, checks against the variable–stack do not always follow this convention.

There are four stacks used by the semantic checker: the type-stack, the variable–stack, the definition-stack, and the and–stack. All but the type–stack are implemented as linked lists. This implementation sheds the disadvantage of static length arrays at the cost of a slight increase in memory and temporal resources. The type–stack uses a fixed-length array of 300 entries because 1) the basic types of real, boolean, integer, natural, and trivial will be accessed most frequently, because they are the building blocks of every type and sequence, and because they can be more easily accessed from an array than from a linked list, 2) a list of 300 type entries should not impose an extreme burden on the programmer, and 3) the planned implementation of sequences will be more straightforward if the type–stack is an array.

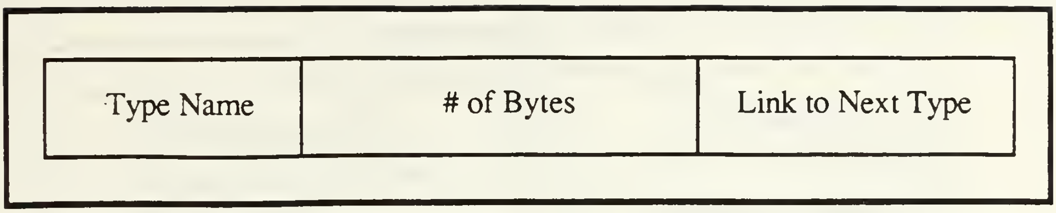


Figure 3.5
Type–Stack Entry

The type–stack, Figure 3.5, is meant to hold both the basic type definitions and user defined type definitions. This stack holds both the name of the type and the number of bytes needed in memory to implement the type. At compiler initialization, it contains the five basic types and user defined types are added as they are encountered. The **begin–end** construct of the language (not implemented yet) allows declared types to be visible over a specified range. It is planned to implement this construct by setting a pointer to the top of the stack upon encountering the **begin** node and then popping the stack to that point after both of the node's subtrees have been checked.

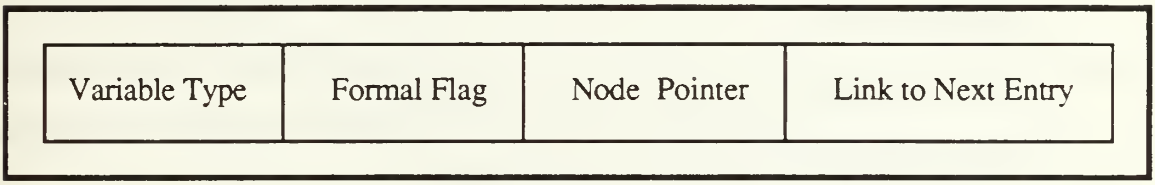


Figure 3.6
Variable–Stack Entry

The variable–stack, Figure 3.6, holds all of the variables, including function names, currently seen by the semantic checker. Each entry holds a pointer to the hash table containing labels, a type, a pointer to the tree node defining it, and a flag to designate whether or not it is a formal. Whenever a variable name is encountered and the name is not a call to a function and not a data definition, it is put into the variable stack. Then, when a scope is exited, the variables local to that scope are dropped from the stack. For example, after a function is defined, all of its formals are popped from the stack.

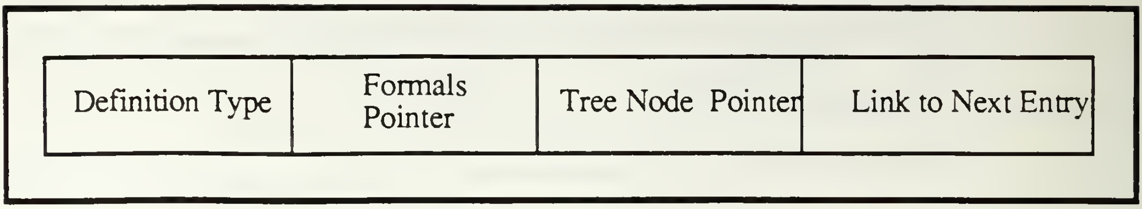
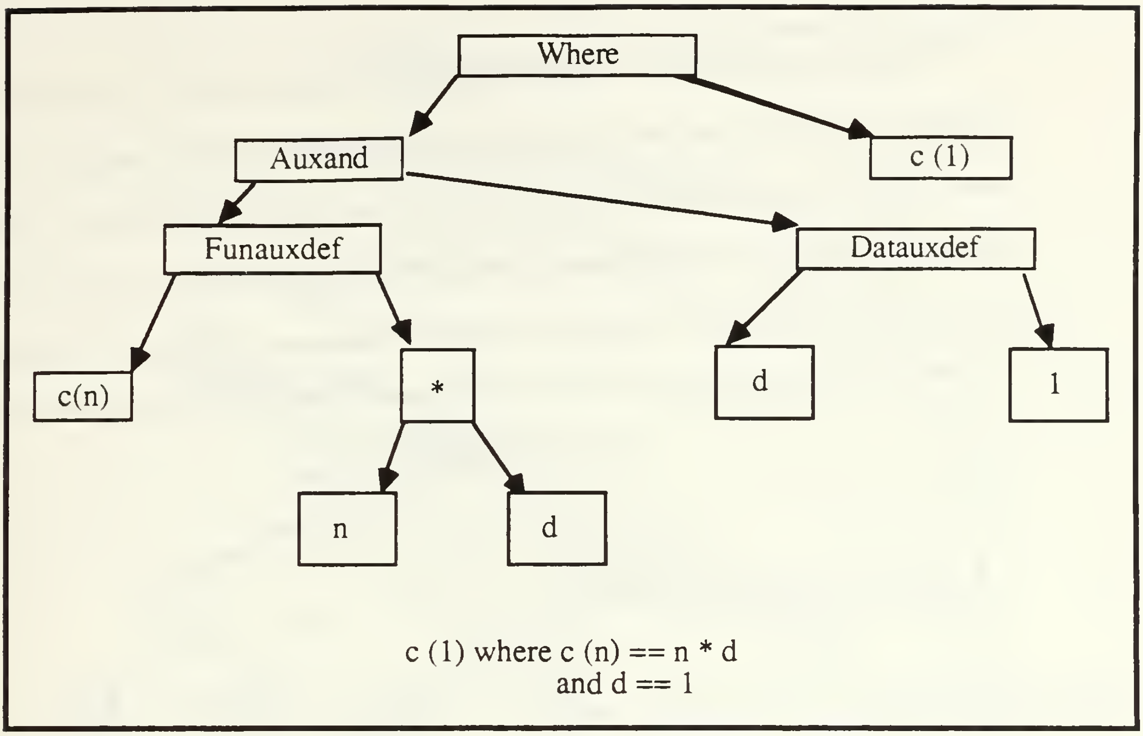


Figure 3.7
Definitions Stack Entry

The definitions–stack, Figure 3.7, contains all of the function and variable definitions visible in a given scope; e.g., the declaration C : $R * $Z -> $B would put the definition C into the definition–stack. This entry would contain the type of C's return value (Boolean), a pointer to the tree node that contains C, and a pointer to a linked list which contains its argument types (Real and Integer). This last field will be null if the declaration is a data definition. This stack grows and shrinks in the same way as the type stack.

The authors considered combining the definitions–stack and the variable–stack because of the similarity between their fields. In fact, one of the primitive implementations was designed in this way. However, this slowed down the search for both definitions and variables considerably, and the overhead needed to implement these two as separate stacks is small: three extra functions and one extra pointer.

The need for the and–stack is derived from the scoping rules imposed by the AND construct. This construct allows a variable to be referenced before it is declared without the benefit of Pascal's forward declaration or equivalent. This is true of other constructs in PHI such as the WHERE construct. However, the AND construct cannot be parsed in such a way that the semantic checker can see all variables before they are used, because either subtree of the AND statement can define variables used by the other subtree. So, a program such as the one depicted in Figure 3.8 needs a vehicle by which it can detect that the variable **d** is defined later in the program. The and–stack is such a vehicle.



```
c (1) where c (n) == n * d
            and d == 1
```

Figure 3.8
Tree With Forward Variables

When the semantic checker reaches the AUXAND node, Figure 3.8, a flag is set to indicate that AUXAND has been traversed, and a pointer is set to the top entry of the and–stack. "Notfound" is returned from the **semcheck** function when the variable **d** is reached, but, since the AND condition has been set, a pointer to **d** is put in the and–stack. Note that **d** is later defined in a data definition (DATAUXDEF node), and when both the left and right subtrees of AUXAND have been checked, all variables in the and–stack are checked against variables in the variable–stack. If a match is found, **d** is defined and removed from the and–stack. In the event that a variable is not found when the AUXAND node's complete subtree has been checked, an error condition (UNDEFINED VARIABLE) would be set. The semantic checker would recognize this condition because the top of the and–stack would not be equal to the mark placed at the top of the stack when the AUXAND node was entered. Nested AUXANDS are possible, but they pose no problem because the top of the and–stack is marked when the auxand node is traversed.

Variables and functions are represented in the run–time by a call to an assembly language subroutine, and each subroutine must have a discrete name. Also, there are several labels found throughout the program, and each of these must have a name. These names are generated by the "name" function found in the **sem_u.c** module. Each name begins with the letter "a", followed by 6 digits. Examples can be seen in Figure 3.4.

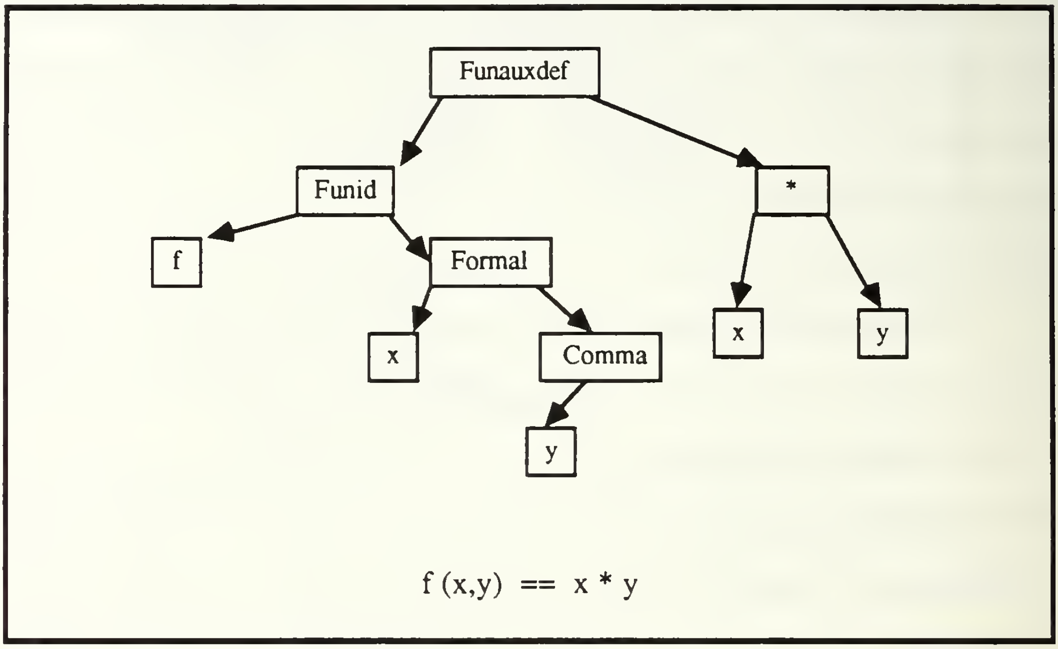


Figure 3. 9
Tree for Function f

Function definitions presented a problem that was solved with a deviation from pure top–down semantic checking. When a function definition (FUNAUXDEF in Figure 3.9) is encountered by semantic checker, the following procedure would be followed (see Figure 3.10 for the function definition entry):

funid node:

```
check for definition-stack entry for "f"
    if not found
        return (ERROR)
get a pointer to the first formal of f
get a pointer to the first formal of definitions-stack entry

while both pointers <> Nil do
```

```
            put variable in varstack; use type pointed to by the formal list
            advance both pointers
    end while loop

    if not (both pointers == nil)
            return (FORMALS MISMATCH)
    else
            put "f" in the variable–stack
            return (Type of f = INTEGER)
    end else
end.
```

funauxdef_node:

```
    left type = semcheck (Left Child)
    right type = semcheck (Right Child)

    if (left type <> right type)
            call a procedure which will either
            convert the right type to the left type or set  an error flag.
    endif
end.
```

When a function is called with arguments, a similar process takes place (refer to Figure 3.11):

actualist : Input is a pointer to the actualist node
Output is error condition

```
    Check definitions-stack for "f"
    if "f" not found
            set error (FUNCTION DEFINITION NOT FOUND)

    set elistptr to first element of element list

    elist (elistptr)

    check var stack for "f"
            if found,
                    generate code to call "f"
            if not found
                    if and_flag = TRUE
                            put "f" in the  and stack
                    else
                            set error (FUNCTION NOT DEFINED)
end.
```

elist: Input is a pointer to the element list node

```
    if pointer->rptr <> nil
            elist (pointer->rptr)

    check type of element against corresponding formal type
    if types don't match
            set error (IMPROPER ARGUMENT TYPE)
            else
                        generate code to put pointers to argument values on the run-time stack
        end.
```

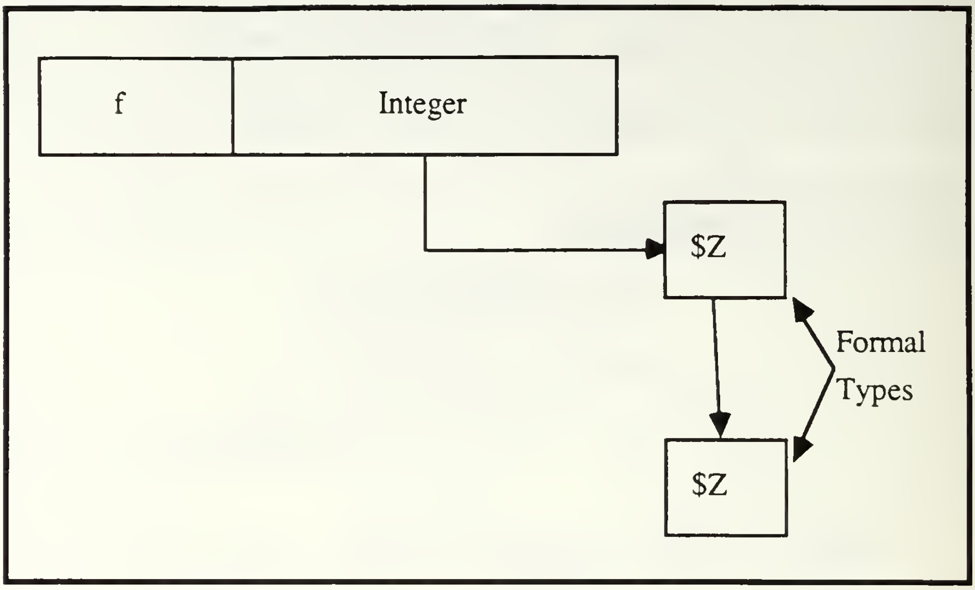


Figure 3.10
Definitions–Table Entry For Function f

Type conversions are implemented in the semantic checker, albeit the code generator does not yet support this feature. The function **hnumconvert** (half number–convert, found in the module **sem0**) will check to see if a conversion of the right subtree of a node to the left subtree type should be accomplished. This is useful for function definitions, where the body of the function may be converted to the type the function returns, but the converse is not acceptable. In addition, the function **numconvert** (found in the **sem0** module) will convert either the left tree type or the right tree type of a node. This is useful for certain arithmetic operations. The semantic checker considers integer–to–real and natural–to–real conversions to be legal. Natural to integer conversions are not implicitly done, since both of these types are represented in exactly the same way. On the other hand, an attempt to return an integer value for a function which has a declared type of natural will result in an error.

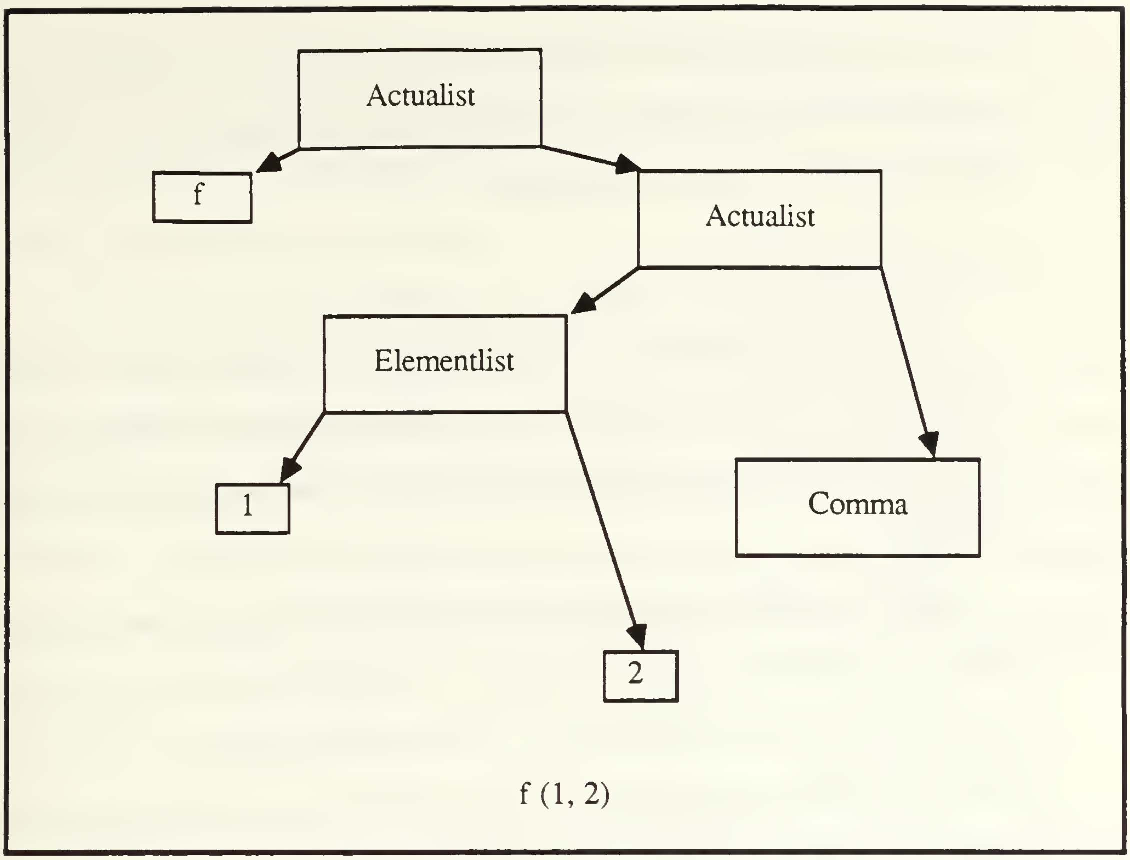


Figure 3.11
Tree for Function Call

Variables of simple type (i.e, natural, integer, or real) need not be declared before use, although such a declaration may be made. If a variable is undeclared when defined by a data definition, the semantic checker will attempt to classify it. If the semantic checker expects to find a boolean value, the variable is easily classified as a boolean and an entry is put into the variable table. If a numeric variable is expected, the semantic checker will try to type it as an integer; failing this, it will be classified as a real number. However, the AND construct alters this somewhat. If a variable is used before it is defined by a data definition, it **must** have been defined using the LETDEF construct.

As noted in the section on run–time, some thought was given to generating all functions and data definitions to one buffer and the "main" program which calls these functions to another buffer. However, this would be an inefficient use of memory space, since one buffer might run out of space while the other is under–utilized. Although there is a proliferation of jump calls in the output using one buffer, an optimizer could easily eliminate all but one call, as noted above.

D. OPTIMIZATION

There is no optimization module implemented in the PHI compiler. In this section an attempt will be made to identify three types of optimization which are suitable for implementation. Also, a small dissertation on what optimizations should not be considered is included.

The first suitable type of optimization is constant folding. The purpose of constant folding is to eliminate multiple consecutive constants in arithmetic expressions [Ref 3:p. 612], and the function **numconvert** in module **sem0** makes an excellent structure in which to implement this optimization. This is because most arithmetic operations call this function. It would be straightforward to put a function that tests the left and right children of an operand node to see if they are constants, then perform the operation in the compiler and generate code for a constant call. However, since the division operators do not call **numconvert**, the constant folding function would have to be inserted in **idiv** and **rdiv** also.

The other two optimizations are post–code generation optimizations. The first one considered is jump optimization. This should be the most worthwhile to implement: if the number of functions and data definitions is n, n > 0, there will be n - 1 unnecessary unconditional jump statements and labels.

These jump statements can be eliminated by replacing the first "jmp" statement with a jump to the last label in the code; then, because "jmp" is not used for anything except to circumnavigate functions and data definitions, all other unconditional jumps and their labels can be eliminated.

The last type of optimization is a form of peephole optimization. Occasionally, there will be a "call ppush" statement followed by a "call ppop" statement. This is unnecessary, and can be eliminated. The 8086 assembly code equivalent of "push" followed by "pop" should not occur in the present design.

Dead code optimization eliminates code inside a jump when that code contains no labels. It is not necessary to implement this type of optimization with the current design, since unconditional jumps are only used to bracket functions and definitions. However, if one accepts the premise that programmers occasionally make mistakes, it might be worthwhile to keep track of which functions are called and eliminate code for those which are not. A message to the programmer concerning this circumstance would be useful, too.

IV. RESULTS & CONCLUSIONS

A. RESULTS

The implementation described in this study demonstrates the design and implementation of a compiler for the functional programming language PHI. Since this implementation is a prototype, it does not possess all of the qualities desirable in a full implementation. However, the necessary hooks are present and the design is mature enough to allow expanding the prototype to a full implementation.

The PHI compiler front–end implements machine independent lexical and syntactic analyzers. This implementation is complete and faithfully follows the syntax of PHI — based on the design of the language as of 07 January 1987. In deciding which modules to include in the front–end and back–end, the authors were originally guided by the traditional methodology of placing the analysis functions in the front–end and generative functions in the back–end [Ref. 8:p. 20]. However, as the design of the PHI compiler progressed, the authors removed semantic analysis from the front–end and combined it with code generation. This produced a one–pass semantic analysis/code generation phase.

The PHI compiler back–end implements a machine dependent one–pass semantic analyzer and Intel 8086 code generator. The semantic analyzer implements the basic constructs of PHI: functions and data definitions may be defined, and the integer, natural number, real number, and boolean types are fully implemented. Implementation of code generation is congruent to that of the semantic analyzer, with the exception that the real number data type has not been implemented.

B. CONCLUSIONS

It is possible, using traditional technologies to design and implement a compiler for the functional programming language PHI. It is not possible to utilize either pure recursive descent or pure deterministic techniques for this implementation. The syntax/semantics of the language forced a degree of non–determinism, and one instance of back–tracking was required in the PHI compiler front–end.

The overall design is highly modularized facilitating the understanding of concept and implementation. The authors think that this approach will greatly reduce maintenance costs and provide greater flexibility in making changes and additions to the PHI programming environment. It should be possible, for example, to use the front–end described in this thesis to drive a PHI interpreter. Being able to abstract out this front–end and use it without change should make the implementation of a PHI interpreter relatively simple. Modularizing the design also increases portability of the compiler to other machines. To demonstrate portability, the authors recompiled the front–end and executed it on a 68000 based processor. This was accomplished with no modifications to the source program, just replacement of C run–time header files for the new target machine.

Removing the semantic analyzer from the front–end permitted coupling semantic analysis with code generation. The fixed–length buffer design of the code generator is suitable for this prototype implementation but should be redesigned utilizing dynamic buffer allocation methods in follow on implementations. The authors think that utilizing a single pass through the parse tree is practical for the basic constructs of PHI and believe this methodology is suitable for future designs of the PHI compiler.

V. FURTHER RESEARCH

Further research may be broken down into two major areas: short and long range projects. The former may be further broken down into two main areas: adding unimplemented features and improving the PHI programming environment. On the other hand, all long-range projects involve only the programming environment. All of these areas are discussed below.

In the prototype of the PHI compiler, both Real and Compound variable types remain unimplemented. Compound variable types consist of sequences, the Trivial type, user defined types, and tuples. Although all of these are recognized by the parser, the semantic checker will not recognize complex types and no code will be generated. The Real type is recognized by the semantic checker, which can discern if conversion from an integer or natural type should be accomplished; however, no code is generated to implement this type in the run-time structures. Note also that operators which operate solely on complex types and reals (e.g., the real divide and concatenate operators) are not implemented.

One other operator not implemented is the "|->" operator. In addition, argument bindings, functionals, and FILEs are not recognized by either the semantic checker or the code generator.

Short–range improvements to the PHI environment may come either after a full implementation is accomplished or may be developed concurrently with the full implementation. Admittedly, the current environment is analogous to instrumentation on a helicopter: there is just enough to know that the system is running! The environment could be improved by implementing the interactive mode of PHI, as opposed to the current batch mode. A sample interactive session of PHI may be found in [Ref. 1:pp 1-17]. Also, an interpreter would be a good starting point toward developing a practical, working environment for PHI. As noted above, the front end of the prototype compiler may be adapted for this purpose; alternatively, due to the structual similarities between PHI and LISP, an ambitious researcher may wish to write an interpreter in LISP.

One final short–range improvement which is not covered by either category would be to allow more than 64K of run–time memory. It would be worthwhile to take advantage of the large amount of memory most modern microcomputers have, especially since sequences and recursion, upon which PHI is based, gobbles up memory with abandon.

When the PHI compiler becomes a serious user's tool, some long–range research will become viable. Sophisticated input and output would be a vital consideration, and the minimal I/O methods now in use would need substantial improvement. The most ambitious researchers in this direction should consider a bit-mapped display with the possibility of a syntax–directed editor. Also, based on the authors' limited experience in PHI programming, a debugger would be a necessary tool for the serious programmer.

LIST OF REFERENCES

1. MacLennan, B. J., *Functional Programming Methodology: Practice and Theory*, to be published by Addison–Wesley, 1987.

2. Horowitz, E. and Munson, J. B., "An Expansive View of Reusable Software," *IEEE Transactions on Software Engineering*, vol. SE–10, No. 5, September 1985.

3. Tremblay, J. and Sorenson, P. G., *The Theory and Practice of Compiler Writing*, McGraw–Hill Book Company, 1985.

4. MacLennan, B. J., *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart and Winston, 1983.

5. Bellot, P., "High Order Programming in Extended FP," in *Lecture Notes in Computer Science*, vol. 201: "Functional Programming Languages and Computer Architecture," edited by Jean–Pierre Jouannaud, Springer–Verlag, Berlin Heidelberg, 1985.

6. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, vol. 21, No. 8, August 1978.

7. Clark, C. and Peyton Jones, S. L., "Strictness Analysis — a Practical Approach," in *Lecture Notes in Computer Science*, vol. 201: "Functional Programming Languages and Computer Architecture," edited by Jean–Pierre Jouannaud, Springer–Verlag, Berlin Heidelberg, 1985.

8. Aho, A. V., Sethi, R., Ullman, J. D., *Compilers Principles, Techniques, and Tools*, Addison–Wesley Publishing Company, 1986.

9. Calingaert, P., *Assemblers, Compilers, and Program Translation*, Computer Science Press, 1979.

10. Knuth, D. E., *Sorting and Searching, The Art of Programming, Vol. 3*, Addison–Wesley Publishing Company, 1973.

11. McKeeman, W. M., "Compiler Construction: An Advanced Course," in *Lecture Notes in Computer Science*, edited by Goos and Hartmanis, Springer–Verlag, New York, 1974.

12. Horowitz, E. and Sahni, S., *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

APPENDIX A

THE FUNCTIONAL LANGUAGE PHI — Φ

(CONCRETE SYNTAX OF Φ — 10/16/86)

GRAMMATICAL NOTATION:

Both '$\{C_1, C_2, \ldots, C_n\}$' and $\left\{\begin{array}{c} C_1 \\ C_2 \\ \vdots \\ C_n \end{array}\right\}$ mean exactly one of $C_1, C_2, \ldots, C_n$.

Similarly, '$[C_1 \mid \ldots \mid C_n]$' and $\left[\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array}\right]$ mean *at most one* of $C_1, \ldots, C_n$. The notation 'C^*' means *zero* or more Cs; 'C^+' means *one* or more Cs; 'CD ...' means a list of one or more Cs separated by Ds. Terminal symbols are quoted when they could be confused with metasymbols.

Grammar:

$$\text{BLOCKBODY} \quad = \quad \left\{\begin{array}{l} \text{QUALEXP} \\ \textbf{LET}\ \text{DEFS ; BLOCKBODY} \end{array}\right\}$$

$$\text{DEF} \quad = \quad \left\{\begin{array}{l} \text{[ID] FORMALS} \equiv \text{QUALEXP} \\ \text{ID : TYPEEXP} \\ \textbf{TYPE}\ \text{ID [FORMALS]} \equiv \text{TYPEEXP} \end{array}\right\}$$

$$\text{QUALEXP} \quad = \quad \left\{\begin{array}{l} \text{EXPRESSION} \\ \text{QUALEXP}\ \textbf{WHERE}\ \text{AUXDEFS} \end{array}\right\}$$

AUXDEFS = AUXDEF **AND** ...

AUXDEF = [ID] FORMALS ≡ EXPRESSION

$$\text{FORMALS} \quad = \quad \left\{\begin{array}{l} \text{ID} \\ \text{(FORMALS, ...)} \end{array}\right\}$$

EXPRESSION = [EXPRESSION ∨] CONJUNCTION

CONJUNCTION = [CONJUNCTION ∧] NEGATION

NEGATION = [~] RELATION

RELATION = [SIMPLEXP RELATOR] SIMPLEXP

RELATOR = { = | ≠ | > | < | ≤ | ≥ | ∈ | ∉ }

SIMPLEXP = [SIMPLEXEP ADDOP] TERM

ADDOP = { + | - | : | ^ }

TERM = [TERM MULOP] FACTOR

MULOP = { X | / | ÷ }

FACTOR = $\left[\begin{array}{c} + \\ - \end{array}\right]$ primary

PRIMARY = $\left\{\begin{array}{l} \text{APPLICATION} \\ \text{PRIMARY}_{\text{APPLICATION}} \end{array}\right\}$

APPLICATION = [APPLICATION] ACTUAL

ACTUAL = $\left\{\begin{array}{l} \text{ID} \\ \text{DENOTATION} \\ \text{CONDITIONAL} \\ \text{COMPOUND} \\ \text{ARGBINDING} \\ \text{BLOCK} \\ \textbf{FILE}\ \text{'}\,\text{CHAR}^{+}\,\text{'} \end{array}\right\}$

DENOTATION = $\left\{\begin{array}{l} \text{'}\,\text{CHAR}^{*}\,\text{'} \\ \text{DIGIT}^{+}\ [.\,\text{DIGIT}^{+}\ ] \\ \text{FORMALS} \mapsto \text{ACTUAL} \end{array}\right\}$

CONDITIONAL = **IF** ARM **ELSIF** ... [**ELSE** EXPRESSION] **ENDIF**

ARM = EXPRESSION **THEN** EXPRESSION

COMPOUND = $\left\{\begin{array}{l} (\ \text{ELEMENTS}\ ) \\ \text{'\{'}\ \text{ELEMENTS}\ \text{'\}'} \\ <\ \text{ELEMENTS}\ > \end{array}\right\}$

ELEMENTS = [QUALEXP , ...]

ARGBINDING = '[' $\left\{\begin{array}{l} \text{OP} \\ \text{OP QUALEXP} \\ \text{QUALEXP OP} \end{array}\right\}$ ']'

OP = { , | RELATOR | ADDOP | MULOP | ! }

BLOCK = **BEGIN** BLOCKBODY **END**

DEFS = DEF **AND** ...

TYPEEXP = TYPEDOM [→ TYPEEXP]

TYPEDOM = TYPETERM [+ TYPEDOM]

TYPETERM = TYPEFAC [**X** TYPETERM]

TYPEFAC = $\left\{\begin{array}{l}\text{TYPEPRIMARY}\\ \text{TYPEPRIMARY}^{*}\\ \text{ID} << \text{TYPEEXP}, \ldots >>\end{array}\right\}$

TYPEPRIMARY = $\left\{\begin{array}{l}\text{ID}\\ \text{PRIMTYPE}\\ (\ \text{TYPEEXP}\ )\end{array}\right\}$

PRIMTYPE = $\{\ \mathbb{R} \mid \mathbb{Z} \mid \mathbb{N} \mid \mathbb{B} \mid \mathbf{1} \mid \textbf{TYPE}\ \}$

For batch use, a program is considered a BLOCKBODY; for interactive use it is considered a SESSION:

SESSION = COMMAND^{+}

COMMAND = $\left\{\begin{array}{l}\text{DEF}\\ \text{QUALEXP}\end{array}\right\}$;

APPENDIX B

THE FUNCTIONAL LANGUAGE PHI — Φ

(CONCRETE SYNTAX OF Φ — 03/03/87)

GRAMMATICAL NOTATION:

Both '$\{C_1,C_2,\ldots,C_n\}$' and $\left\{\begin{array}{c} C_1 \\ C_2 \\ \vdots \\ C_n \end{array}\right\}$ mean exactly one of C_1, C_2,..., C_n.

Similarly, '$[C_1 \mid \ldots \mid C_n]$' and $\left[\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array}\right]$ mean *at most one* of C_1,..., C_n. The notation 'C^*' means *zero* or more Cs; 'C^+' means *one* or more Cs; 'CD ...' means a list of one or more Cs separated by Ds. Terminal symbols are quoted when they could be confused with metasymbols.

Grammar:

BLOCKBODY = $\left\{\begin{array}{l} \text{QUALEXP} \\ \textbf{LET}\text{ DEFS ; BLOCKBODY} \end{array}\right\}$

DEF = $[\textbf{REC}]\left\{\begin{array}{l} \text{[ID, ... : TYPEEXP \{}\textbf{BE}\mid\textbf{IS}\text{ \}] [ID] FORMALS} \equiv \text{QUALEXP} \\ \textbf{TYPE}\text{ ID [FORMALS]} \equiv \text{TYPEEXP} \end{array}\right\}$

QUALEXP = $\left\{\begin{array}{l} \text{EXPRESSION} \\ \text{QUALEXP }\textbf{WHERE}\text{ AUXDEFS} \end{array}\right\}$

AUXDEFS = AUXDEF **AND** ...

AUXDEF = [ID] FORMALS ≡ EXPRESSION

FORMALS = $\left\{\begin{array}{l} \text{ID} \\ \text{(FORMALS, ...)} \end{array}\right\}$

EXPRESSION = [EXPRESSION $\vee$] CONJUNCTION

CONJUNCTION = [CONJUNCTION $\wedge$] NEGATION

NEGATION = [~] RELATION

RELATION = [SIMPLEXP RELATOR] SIMPLEXP

RELATOR = $\{ = \mid \neq \mid > \mid < \mid \leq \mid \geq \mid \in \mid \notin \mid \rightarrow \}$

SIMPLEXP = [SIMPLEXEP ADDOP] TERM

ADDOP = { + | - | : | ^ | + | '|'}

TERM = [TERM MULOP] FACTOR

MULOP = { X | / | ÷ | ∘ | ; | **X** }

$$\text{FACTOR} = \left[\begin{array}{c} + \\ - \end{array}\right] \text{PRIMARY}$$

$$\text{PRIMARY} = \left\{\begin{array}{l} \text{APPLICATION} \\ \text{PRIMARY}_{\text{APPLICATION}} \end{array}\right\}$$

APPLICATION = [APPLICATION] ACTUAL

$$\text{ACTUAL} = \left\{\begin{array}{l} \text{ID [« TYPEEXP, ...»]} \\ \text{DENOTATION} \\ \text{CONDITIONAL} \\ \text{COMPOUND} \\ \text{ARGBINDING} \\ \text{BLOCK} \\ \{\ \textbf{FILE} \mid \textbf{STREAM}\ \}\text{' CHAR}^{+}\text{ '} \end{array}\right\}$$

$$\text{DENOTATION} = \left\{\begin{array}{l} \text{'CHAR}^{*}\text{'} \\ \text{DIGIT}^{+}\ [.\ \text{DIGIT}^{+}\ ] \\ \textbf{NIL} \\ \text{FORMALS} \mapsto \text{ACTUAL} \end{array}\right\}$$

CONDITIONAL = **IF** ARM **ELSIF** ... [**ELSE** EXPRESSION] **ENDIF**

ARM = EXPRESSION **THEN** EXPRESSION

$$\text{COMPOUND} = \left\{\begin{array}{l} \text{'[' ELEMENTS ']'} \\ (\ \text{ELEMENTS}\ ) \\ \text{'\{' ELEMENTS '\}'} \\ <\ \text{ELEMENTS}\ > \end{array}\right\}$$

ELEMENTS = [EXPRESSION , ...]

$$\text{ARGBINDING} = \text{'['} \left\{\begin{array}{l} \text{OP} \\ \text{OP ACTUAL} \\ \text{ACTUAL OP} \end{array}\right\} \text{']'}$$

OP = { , | RELATOR | ADDOP | MULOP | **SUB** }

BLOCK = BEGIN BLOCKBODY **END**

DEFS = DEF **AND** ...

TYPEEXP = TYPEDOM [→ TYPEEXP]

TYPEDOM = TYPETERM [+ TYPEDOM]

TYPETERM = TYPEFAC [**X** TYPETERM]

TYPEFAC = $\left\{ \begin{array}{l} \text{TYPEPRIMARY}^{*} \\ \text{TYPEPRIMARY [ACTUAL]} \end{array} \right\}$

TYPEPRIMARY = $\left\{ \begin{array}{l} \text{ID [« TYPEEXP, ... »]} \\ \text{PRIMTYPE} \\ \text{(TYPEEXP)} \end{array} \right\}$

PRIMTYPE = $\{ \mathbb{R} \mid \mathbb{Z} \mid \mathbb{N} \mid \mathbb{B} \mid \mathbf{1} \mid \textbf{TYPE} \}$

For batch use, a program is considered a BLOCKBODY; for interactive use it is considered a SESSION:

SESSION = COMMAND^{+}

COMMAND = $\left\{ \begin{array}{l} \text{LET DEF} \\ \text{QUALEXP} \end{array} \right\}$;

APPENDIX C
ASCII REPRESENTATION OF — Φ

Reference	ASCII
≡	==
<	LESS
≤	<=
>	>=
≠	<>
∈	IN
∉	NOTIN
∨	\/
∧	/\
~	~
X	*
/	/
÷	%
→	->
^	^
↦	|->
A_i	A ! i
T^*	T @
ℝ	$R
ℤ	$Z
ℕ	$N
𝔹	$B
𝟙	$1

APPENDIX D

THE FUNCTIONAL LANGUAGE—Φ

(RIGHT–RECURSIVE GRAMMAR)

Note: $(\ldots)^*$ means zero or more occurrences
$(\ldots)^+$ means one or more occurrences
$(\ldots)^n$ means from zero to n occurrences
(x | y) means either x or y, but not both

BLOCK ::= **BEGIN** BLOCKBODY **END**

BLOCKBODY ::= **LET** DEFS; BLOCKBODY
QUALEXP

DEFS ::= DEF (**AND** DEFS)*

DEF ::= (ID)^1 FORMALS ≡ QUALEXP
ID : TYPEEXP
TYPE ID (FORMALS)^1 ≡ TYPEEXP

QUALEXP ::= EXPRESSION (**WHERE** AUXDEFS)*

AUXDEFS ::= AUXDEF (**AND** AUXDEF)*

AUXDEF ::= (ID)^1 FORMALS ≡ EXPRESSION

FORMALS ::= (FORMALS (,FORMALS)*)
ID

EXPRESSION ::= CONJUNCTION ($\vee$ CONJUNCTION)*

CONJUNCTION ::= NEGATION($\wedge$ NEGATION)*

NEGATION ::= (~)^1 RELATION

RELATION ::= SIMPLEXEP (RELATOR SIMPLEXP)^1

```
RELATOR         ::= =
                    ≠
                    LESS
                    GREATER
                    ≤
                    ≥
                    ∈
                    ∉
SIMPLEXP        ::= TERM (ADDOP TERM)*
ADDOP           ::= +
                    -
                    :
                    ^
TERM            ::= FACTOR (MULOP FACTOR)*
MULOP           ::= *
                    /
                    ÷
FACTOR          :=  + PRIMARY
                    - PRIMARY
                    PRIMARY
PRIMARY         ::= APPLICATION (! APPLICATION)*
APPLICATION     ::=  (ACTUAL)+
ACTUAL          ::= ID
                    DENOTATION
                    CONDITIONAL
                    COMPOUND
                    ARGBINDING
                    BLOCK
                    FILE '(CHAR)+ '          Note: CHAR can =  ASCII 32 ... ASCII 126
DENOTATION      ::= '(CHAR)* '               Note: CHAR can =  ASCII 32 ... ASCII 126
                    (DIGIT)+                 Note: DIGIT can =  0 ... 9
                    (DIGIT)+ . (DIGIT)+
                    FORMALS |→ ACTUAL
ID              ::= ALF (ALFNUM)*            Note: ALF can =  a...z, A...Z
                                             ALFNUM can =  a...z, A...Z, 0...9, _
CONDITIONAL     ::= IF ARM (ELSIF ARM)* (ELSE EXPRESSION)1 ENDIF
ARM             ::= EXPRESSION THEN EXPRESSION
```

COMPOUND ::= ((ELEMENTS)^1)
{ (ELEMENTS)^1 }
< (ELEMENTS)^1 >

ELEMENTS ::= QUALEXP(,QUALEXP)*

ARGBINDING ::= [op]
[OP QUALEXP]
[QUALEXP OP]

OP ::= ,
RELATOR
ADDOP
MULOP
!

TYPEEXP ::= TYPEDOM (→ TYPEDOM)*

TYPEDOM ::= TYPETERM (+ TYPETERM)*

TYPETERM ::= TYPEFAC (* TYPEFAC)*

TYPEFAC ::= TYPEPRIMARY$^@$
TYPEPRIMARY
ID <<TYPEEXP (,TYPEEXP)* >>

TYPEPRIMARY ::= (TYPEEXP)
ID
PRIMTYPE

PRIMTYPE ::= $\mathbb{R}$
$\mathbb{Z}$
$\mathbb{N}$
$\mathbb{B}$
1
TYPE

FOR INTERACTIVE IMPLEMENTATION OF Φ

SESSION ::= (COMMAND)$^+$

COMMAND ::= (DEF | QUALEXP) ;

APPENDIX E

ROCK COMPILER HEADER FILES

```
/**************************************************************************
*    THIS FILE CONTAINS HEADER FILES REQUIRED BY THE ROCK COMPILER        *
**************************************************************************/

/**************************************************************************
*                              PUBLIC DOMAIN SOFTWARE                     *
                                                                          *
* Name       :   scanner definitions                                      *
* File       :   scanner.h                                                *
* Authors    :   Maj E.J. COLE / Capt J.E. CONNELL                        *
* Started    :   10/10/86                                                 *
* Archived   :   12/11/86                                                 *
* Modified   :   01/10/87 - Update keywords   JC                          *
***************************************************************************
* This file contains definitions used by the scanner,parser, and          *
* error recovery routines.                                                *
***************************************************************************
* Modified    :  01/10/87 Corrections to comply with latest definitions *
*                of the language and update keywords. JC                  *
**************************************************************************/
#ifndef   EOF_

#define EOF_           -2
#define FALSE           0
#define TRUE            1
#define BYTENUM         2                        /* system dependent - sizeof(int) */
#define MAX_KEYWORDS   17                        /* really 18, ranges from 0 - 17  */
#define NAMESIZE       18                        /* length of str, 16 chars + '\0' */
#define MAXLINE        80
#define TABLESIZE     107                        /* hash const/size of name array  */

                       /* General Token   Types */
            /* Listing of symbols can be found at end of list */

#define EOLN_           3
#define LEQ_            4
#define NEQ_            5
#define ST_SEQUENCE_    6
#define GEQ_            7
#define END_SEQUENCE_   8
#define EQ_             9
#define ADD_           10
#define SUB_           11
#define MULT_          12
#define IDIV_          13
#define RDIV_          14
#define SEMI_          15
#define SUBSCRIPT_     16
```

```
#define COMMA_          17
#define LTPAREN_        18
#define RTPAREN_        19
#define EQUIV_          20
#define ORLOG_          21
#define ANDLOG_         22
#define NEGLOG_         23
#define COLON_          24
#define CAT_            25
#define LTBRAKET_       26
#define RTBRAKET_       27
#define LTSQUIG_        28
#define RTSQUIG_        29
#define EMPT_LIT_       30
#define RTARROW_        31
#define LINERTARROW_    32
#define LITERAL_        33
#define IDENTIFIER_     34
#define CONSTANT_       35
#define REAL_           36
#define INTEGER_        37
#define NATURAL_        38
#define BOOLEAN_        39
#define TRIVIAL_        40
#define CHAR_           41
#define STRING_         42
#define STAR_           43
#define POS_            44
#define NEG_            45
#define KW_             46                            /* KEYWORD                         */

/* eof, error, unknown token, <=, <>, <, >=, >, =, +, -, *, %, /, ;, !,
,, (, ), ==, \/, /\, ~, :, ^, [, ], {, }, '', ->, |->, literal,
identifier, constant, $R, $Z, $N, $B,$1, character, string, @,
unary plus, unary minus, keyword                                      */


                                   /* Keywords */
#define AND_              0
#define BEGIN_            1
#define ELSE_             2
#define ELSIF_            3
#define END_              4
#define ENDIF_            5
#define FILE_             6
#define GREATER_          7
#define IF_               8
#define IN_               9
#define LESS_            10
#define LET_             11
#define NOTIN_           12
#define READ_            13
#define THEN_            14
#define TYPE_            15
#define WHERE_           16
#define WRITE_           17
```

```
#define   CALLOC(y,x)   ((x*) calloc(y,sizeof(x)))
struct   NStruct    {                              /* structure to hold names from   */
                                                   /* user prog                      */
   char             name[NAMESIZE];
   struct NStruct   *link;
   };
typedef struct NStruct   NameRec;
extern   char   *calloc();
extern   char   *malloc();

#endif
```

```
/***********************************************************************
*                           PUBLIC DOMAIN SOFTWARE                     *
*                                                                      *
* Name      :   parser definitions                                     *
* File      :   parser.h                                               *
* Authors   :   Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :   10/20/86                                               *
* Archived  :   12/11/86                                               *
* Modified  :   01/12/87 - update NodeStruct definition JC             *
************************************************************************
* This file contains definitions used by the parser                    *
************************************************************************
* Modified   :  01/10/87 - update NodeStruct to hold the type of the   *
*               node                                                   *
***********************************************************************/

#ifndef   LETDEF

#define   LETDEF         71
#define   DEFAND         72
#define   KINDEF         73
#define   FUNID          74
#define   FUNDEF         75
#define   DATADEF        76
#define   TDEFID         77
#define   TDEFFUN        78
#define   DATAAUXDEF     79
#define   FUNAUXDEF      80
#define   AUXAND         81
#define   ACTUALLIST     82
#define   SEQUENCE       83
#define   FORMAL         84
#define   ELLIST         85
#define   EMPTYCOMPOUND  88
#define   EMPTYSEQUENCE  89
#define   ARGBINDOP      90
#define   ARGLEADOP      91
#define   ARGTRAILOP     92
#define   TYPEPLUS       93
#define   TYPETIMES      94
#define   TYPEEXPLIST    95

#define   LEFT            1
#define   RIGHT           2
#define   ERROR_         -1
#define   BUFSIZE       512

typedef   int NodeType;

struct NodeStruct {                                   /* operator node in tree         */
   NodeType     name;                                 /* int defined as the operator   */
   long         index;                                /* pointer to constant,literal,id */
   int          type;                                 /* the type of the node          */
   int          ln;                                   /* line no in source text where  */
                                                      /*  token can be found           */
   char         label [8];                            /* Label used by functions to    */
                                                      /* refer  to code                */
   long         addr;                                 /* Addr of the var or function   */
                                                      /* value in the run time's virtual*/
                                                      /* memory                        */
   struct   NodeStruct   *lptr;                      /* left ptr                        */
   struct   NodeStruct   *rptr;                      /* right ptr                       */
   };
typedef   struct NodeStruct NodeRec, *nodal;

NodeRec   *CreateNode();
char      *NodeName();
                                                     /* global var-list number errors  */
extern int   num_errors;                             /* during scan and parse          */
extern int   argbind;                                /* global flag - used to make PHI */
                                                     /* deterministic                  */

extern char *calloc();                               /* def used from <stdlibs.h>      */
extern char *malloc();
extern ErrorHandler();
extern WriteErrors();

    /****************** External Utility Functions *****************/

extern NodeRec *CreateNode();
extern char    *NodeName();
extern MakeNewRoot();
extern IsFormal();
extern IBall();
extern EatEm();
extern long ByPass();

#include <scanner.h>
#include <errors.h>

#endif
```

```
/***********************************************************************
*                         PUBLIC DOMAIN SOFTWARE                       *
*                                                                      *
* Name      :   error file definitions                                 *
* File      :   erors.h                                                *
* Authors   :   Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :   01/20/87                                               *
* Archived  :   04/07/87                                               *
* Modified  :                                                          *
************************************************************************
* This file contains definitions used by the error recovery routines. *
************************************************************************
* Modified                                                             *
***********************************************************************/

#ifndef   MAXERRORS

#define MAXERRORS  10

/****************************** PARSER ERRORS ***************************************/

#define   ERR0      0                              /* '|' or '|-' w/o '>'            */
#define   ERR1      1                              /* RESERVED FOR FUTURE USE        */
#define   ERR2      2                              /* '\' w/o '/' -- bad logical OR  */
#define   ERR3      3                              /* '$' w/o proper following char  */
#define   ERR4      4                              /* invalid numeric constant       */
#define   ERR5      5                              /* literal w/o ending             */
#define   ERR6      6                              /* unidentified char in input file*/
#define   ERR7      7                              /* out of memory                  */
#define   ERR8      8                              /* error in statement following   */
                                                   /* 'xx'                           */
#define   ERR9      9                              /* error in type definition       */
                                                   /* following 'xx'                 */

#define   ERR_a    10                              /* unable to complete eval of     */
                                                   /* the blockbody                  */
#define   ERR_b    11                              /* missing or misplaced ; after   */
                                                   /* definition                     */
#define   ERR_c    12                              /* invalid QualExp                */
#define   ERR_d    13                              /* invalid TypeExp                */
#define   ERR_e    14                              /* bad or missing formals         */
#define   ERR_f    15                              /* missing or misplaced           */
#define   ERR_g    16                              /* missing ID after 'TYPE'        */
#define   ERR_h    17                              /* bad definition after AND       */
#define   ERR_i    18                              /* missing or bad AuxDef after    */
                                                   /* WHERE                          */
#define   ERR_j    19                              /* missing or misplaced ')'       */
#define   ERR_k    20                              /* error in processing            */
                                                   /* successive Actuals             */
#define   ERR_l    21                              /* missing literal after keyword  */
                                                   /* FILE"                          */
#define   ERR_m    22                              /* missing or invalid exp after   */
                                                   /* keyword ==>                    */
#define   ERR_n    23                              /* IF statement w/o ENDIF         */
#define   ERR_o    24                              /* error in formals preceding |-> */
#define   ERR_p    25                              /* missing or invalid QualExp     */
                                                   /* following comma op             */
#define   ERR_q    26                              /* error in ArgBinding - check    */
                                                   /* QualExp  or ]                  */
#define   ERR_r    27                              /* off in OZONE-unimplemented     */
                                                   /* feature                        */
```

```
#define   ERR_s    28                                      /*                                  */
#define   ERR_t    29                                      /*                                  */
#define   ERR_u    30                                      /*                                  */
#define   ERR_v    31                                      /*                                  */
#define   ERR_w    32                                      /*                                  */
#define   ERR_x    33                                      /*                                  */
#define   ERR_y    34                                      /*                                  */
#define   ERR_z    35                                      /*                                  */

          /* NOTE:  s through z reserved for future use */

/********************* SEMANTIC ERRORS *******************************/

#define   ERR_aa   35                                      /* Numeric value expected           */
#define   ERR_bb   35                                      /* Natural expected                 */
#define   ERR_cc   35                                      /* Integer or natural expected      */
#define   ERR_dd   35                                      /* Error in Tuple Definition        */
#define   ERR_ee   35                                      /* Undefined var in "and" scope     */
#define   ERR_ff   35                                      /* Function w/o function def        */
#define   ERR_gg   35                                      /* Formals mismatch                 */
#define   ERR_hh   35                                      /* Undefined function               */
#define   ERR_ii   35                                      /* Real Number expected             */
#define   ERR_jj   35                                      /* Invalid Constant                 */
#define   ERR_kk   35                                      /* Boolean value Expected           */
#define   ERR_ll   35                                      /* Boolean Operator Expected        */
#define   ERR_mm   35                                      /* Out of run-time memory space     */

#endif
```

```
/***********************************************************************
*                         PUBLIC DOMAIN SOFTWARE                      *
*                                                                     *
* Name      :  Semantic Definitions Header File                       *
* File      :  Semcheck.h                                             *
* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :  01/01/87                                               *
* Archived  :  04/10/87                                               *
* Modified  :  04/13/87  "FILENAME" eliminated  EC                    *
***********************************************************************
* This file contains the header file and definitions for the semantic *
* checker and code generator of the PHI compiler                      *
***********************************************************************
* Modified  :  04/13/87  "FILENAME" eliminated; output path now       *
*               depends on user's input  EC                           *
**********************************************************************/

/****************************** Externals *****************************/
#include <scanner.h>
#include <parser.h>
#include <errors.h>
#include <stdio.h>

/******************************* Globals ******************************/
#define NOTFOUND 0                              /* Definition for findvar        */
#define UNTYPED 0                               /* Type Definitions and sizes    */
#define BOOLEAN 1
#define BOL_BYTES 2
#define REAL 2
#define REAL_BYTES 4
#define INTEGER 3
#define INT_BYTES 2
#define NATURAL 4
#define NAT_BYTES 2

#define ERROR 0
#define MAXADDR 64000                           /* Max # of bytes in var space   */

#define MAXTYPES 300                            /* Max # of types in one scope   */
#define CODE_SIZE 20000                         /* Max size of code buffer       */
#define    START_ADDR 0                         /* Starting address for varspace */
#define    TYPE_INIT  5                         /* Pointer to the last initial   */
                                                /* typetable entry               */
#define    CNTRL_Z    26                        /* Control Z ascii               */
#define    ENDSTRING 0                          /* String terminator             */
#define NUM_BASE 48                             /* Lowest ascii number           */
#define STACKSIZE 10000                         /* Increase in stack size        */
#define SIZEBUFFER 30000                        /* Size of output buffer         */

#define ADD 1                                   /* Sem check codes for arith ops */
#define SUB 2
#define DIVIDE 3
#define MULT 4

#define SEM_ERR  0                              /* Flag to indicate semantic     */
                                                /* error follows                 */

#ifndef NULL
  #define NULL 0
#endif
```

```
/************************************************************************************
*                                                                                   *
*                                 Type Definitions                                  *
*                                                                                   *
************************************************************************************/

typedef int optype,                               /* Arithmetic operations         */
       FLAG,                                      /* Generic flag type             */
       PHITYPE;                                   /* Types found in language       */

typedef char stg [20];                            /* Assembly language code names  */

typedef struct and_struct *and_ptr;               /* Pointer to and_table entries  */

/****************************** Typetable Definitions *******************/
typedef struct typenode {                         /* Typetable entries             */
          char name [10];
          int bytes;
          struct typenode *typeptr;
                } tnode;

/************************* Formallist Definitions *********************/
typedef struct formnode {                         /* Formal stack                  */
                    int name, type;               /* formname, formtype            */
           struct formnode *link;                 /* Link for list                 */
           } fnode;

/************************* Vartable   Definitions *********************/
typedef struct varnode {                          /* Entry for variable stack      */
          int type,                               /* varname, vartype              */
              form,                               /* Flag set if var is a formal   */
              def;                                /* True if var is a definition   */
          nodal nptr;                             /* ptr to defining node          */
          fnode *fptr;                            /* ptr to formals                */
          struct varnode *link;                   /* Link for list                 */
              } *varptr;

/************************* Deftable   Definitions *********************/
typedef struct defnode {
          int type;                               /* varname, vartype              */
          nodal nptr;                             /* ptr to defining node          */
          fnode *fptr;                            /* ptr to formals                */
          struct defnode *link;                   /* Link for list                 */
              } *defptr;

/****************************** And Definitions *************************/
struct and_struct                                 /* Structure for and lists       */
         {nodal ptr;                              /* Ptr to nodal containing var name
   */
          int buffptr;                            /* Ptr to buffer where           *
                                                  /* name is called                */
          struct and_struct *link;                /* Link for linked list          */
         };
```

```
/**********************************************************************
*                         PUBLIC DOMAIN SOFTWARE                      *
*                                                                     *
* Name      : User Header                                             *
* File      : user.h                                                  *
* Authors   : Maj E.J. COLE / Capt J.E. CONNELL                       *
* Started   : 04/01/87                                                *
* Archived  : 04/10/87                                                *
* Modified  :                                                         *
***********************************************************************
* This file is the header file for the user interface module          *
* (user.c)                                                            *
***********************************************************************
* Modified  :                                                         *
**********************************************************************/

/****************************** Globals ******************************/
#define BUFFLENGTH 30                              /* Max size of input file name +  */
                                                   /* directory                      */
#define NOTFOUND 0
#define BSIZE 1000                                 /* Input buffer size              */
#define BLOCKSIZE 50                               /* Input block size               */

#define BACKSPACE 8                                /* ASCII Equivilents              */
#define EOLN 13
#define ESCAPE 27

#define GETPROGRAM "Program to Compile -> "        /* Messages to observer           */
#define HEADER1 "ROCK COMPILER"
#define HEADER2 "Press Escape Key to Exit Compiler"
#define FILE1_ERROR "File not Found"
#define FILE2_ERROR "Press ESCAPE to exit, any other key to continue"
#define WAIT "Compiling: Please Wait"
#define PAUSE "PRESS ANY KEY TO CONTINUE"

#define ERRORFILE "errors.phi"                     /* Textfile of errors             */
```

APPENDIX F

ROCK COMPILER — MAIN MODULE

```
/**********************************************************************
* PUBLIC DOMAIN SOFTWARE                                              *
*                                                                     *
* Name      :  Main Rock Module                                       *
* File      :  Rock_main.c                                            *
* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :  01/06/87                                               *
* Archived  :  04/10/87                                               *
* Modified  :  04/13/87  Output files put to vdisk   EC               *
***********************************************************************
* This file contains the following modules for the PHI compiler:     *
*                                                                     *
*      R_Initial               Semcheck                 Main          *
*                                                                     *
* Algorithm :                                                         *
*    This contains the main procedure for the phi compiler, in add-   *
* ition to the initialization procedure & the main semantic checking  *
* procedure.  The main module inits the program, sets up the screen   *
* by calling "user ()", & decides whether an error routine needs      *
* to be called.  It also closes out the input file.                   *
*    The "semcheck procedure is designed to be called by any function *
* with a ptr to a parse tree node as an argument.  It will then       *
* determine which sub-module is necessary to check the node.          *
*    "R_Initial" presently has the function of initializing the type  *
* table.                                                              *
*                                                                     *
***********************************************************************
* Modified :  04/13/87  Output files written to vdisk, "d:"   EC      *
**********************************************************************/

/**************************** Externals *******************************/

#include <semcheck.h>

extern void c_startup (),                       /* Initializer for code buffer    */
       c_ending (),                             /* Close out for code generator   */
       user (),                                 /* User interface                 */
       user_err (),                             /* Error writing interface        */
       p_close (),                              /* Close source file              */
       set_page (),                             /* Change video display page      */
       mov_cursor ();                           /* Move cursor to specified locat */

extern FLAG err_found;
extern nodal parser ();

/*************************** Globals *******************************/
unsigned _stack = STACKSIZE;
```

```
/*************************** R_Initial ******************************/
    void
r_initial ()                                        /* Initialize semantic checking   */
{extern tnode types [];

   strcpy (types [UNTYPED].name, "untyped");       /* Set up type table              */
   types [UNTYPED].bytes = NULL;
   strcpy (types [BOOLEAN].name, "boolean");
   types [BOOLEAN].bytes = BOL_BYTES;
   strcpy (types [REAL].name, "real");
   types [REAL].bytes = REAL_BYTES;
   strcpy (types [INTEGER].name, "integer");
   types [INTEGER].bytes = INT_BYTES;
   strcpy (types [NATURAL].name, "natural");
   types [NATURAL].bytes = NAT_BYTES;
}

/************************* SemChecker ****************************/
    PHITYPE
semcheck (ptr)                                      /* Breaks Sem Check into cases    */
    nodal ptr;
{extern PHITYPE tkindef (), trtarrow (),
 tfunid (), tid (), tconstant (), tactuallist (), tactuals ();
 PHITYPE type;

   switch (ptr->name) {
     case (ADD_) :
     case (SUB_) :
     case (MULT_) :
     case (RDIV_) :
     case (IDIV_) :
     case (COLON_) :
     case (CAT_) : type = arithop (ptr);
          break;
     case (POS_) :
     case (NEG_) : type = tprimary (ptr);
          break;
     case (ORLOG_) : type = tor (ptr);
          break;
     case (ANDLOG_) : type = tand (ptr);
          break;
     case (NEGLOG_) : type = tnegation (ptr);
          break;
     case (KINDEF) : tkindef (ptr);
          break;
     case (RTARROW_) : type = trtarrow (ptr);
          break;
     case (LETDEF) : tletdef (ptr);
          break;
     case (KW_ + WHERE_) : type = twhere (ptr);
          break;
     case (AUXAND) : tauxand (ptr);
          break;
     case (DATAAUXDEF) : tdatauxdef (ptr);
          break;
     case (FUNAUXDEF) : type = tfunauxdef (ptr);
          break;
     case (FUNID) : type = tfunid (ptr);
          break;
     case (ACTUALLIST) : type = tactuals (ptr);
          break;
     case (COMMA_) :
```

```
     case (ELLIST) : telist (ptr);
          break;
     case (TYPETIMES) : type = ttypetimes (ptr);
          break;
     case (EQ_)   :
     case (LEQ_)   :
     case (NEQ_)   :
     case (GEQ_)   :
     case (KW_ + GREATER_) :
     case (KW_ + LESS_) :
     case (KW_ + IN_) :
     case (KW_ + NOTIN_) : type = tcomp (ptr);
                 break;
     case (KW_ + IF_) : type = tif (ptr);
              break;
     case (KW_ + ELSE_) : type = telse (ptr);
                break;
     case (KW_ + THEN_) : type = tthen (ptr);
                break;
     case (KW_ + ELSIF_) : type = telseif (ptr);
                break;
     case (IDENTIFIER_) : type = tid (ptr);
           break;
     case (CONSTANT_) : type = tconstant (ptr);
           break;
     case (REAL_) : type = REAL;
         break;
     case (INTEGER_) : type = INTEGER;
                  break;
     case (BOOLEAN_) : type = BOOLEAN;
         break;
     case (NATURAL_) : type = NATURAL;
                  break;

     default : terror (ERR_r, ptr->ln);            /* Unimplemented feature found,    */
                                                   /* so sandbag programmer           */
         break;
         }

   return (type);
}

/************************************** Main ***************************************/
main ()
{extern char prefix [];                            /* Prefix of the souce file        */
 extern void curson (), cursoff ();                /* Turn cursor on and off          */
 char name_holder [30];                            /* Holder for prefix name          */
 nodal root;                                       /* Root of the Parse tree          */

   c_startup ();                                   /* Initialize and open files       */
   r_initial ();

   user ();                                        /* User interface                  */

   if (root = parser ())       {                   /* Parse code; continue if root    */
                                                   /* not equal to nil                */
         set_page (2);                             /* Freeze current video display    */
         cursoff ();
        semcheck (root);                           /* Semantic check and code gen     */

         if (!err_found)   {
       c_ending ();                                /* Clean up and close out files    */
```

```
        forkl ("d:rasm86.exe", "d:rasm86.exe",       /* Assemble the code              */
                prefix, NULL);
        forkl ("d:link86.exe", "d:link86.exe",       /* Link object files together     */
               .prefix,",", "d:u", NULL);

        strcpy (name_holder, prefix);
        strcat (name_holder, ".lst");                /* clean up the loose files       */
        remove (name_holder);
        strcpy (name_holder, prefix);
        strcat (name_holder, ".sym");
        remove (name_holder);
        set_page (0);
        curson ();
            }       }

    if (err_found || !root ) {                       /* Print error files if req       */
         set_page (0);
         curson ();
         user_err ();
               }
    p_close ();                                      /* Close source file              */

    execl ("rock.exe", "rock.exe", NULL);            /* Execute rock again; exit comes */
                                                     /* from inside main Procedure     */
}
```

APPENDIX G

ROCK COMPILER — SCANNER

```
/***********************************************************************
*                          PUBLIC DOMAIN SOFTWARE                     *
* Name      :   Scanner                                               *
* File      :   Scan2.c                                               *
* Authors   :   Maj E.J. COLE / Capt J.E. CONNELL                     *
* Started   :   10/10/86                                              *
* Archived  :   12/11/86                                              *
* Modified  :   04/23/87 tokens no longer output to intermediate file. *
***********************************************************************
* This file contains the execution modules for the scanner:           *
*                                                                     *
*                 GetToken(),         IsKeyWord()                     *
*                                                                     *
* Algorithm :  GetToken is called from FillBuffer() and returns an    *
*              integer code to uniquely identify the token.           *
*              IsKeyWord() checks each identifier to insure it's not  *
*              a PHI KeyWord.                                         *
***********************************************************************
* Modified   :  01/10/87 Corrections to comply with latest definitions *
*               of the language. JC                                   *
*            :  01/10/87 GetToken() returns CONSTANT_ vice REAL_ or   *
*               INTEGER_. JC                                          *
*            :  01/21/87 Error Recovery added and files combined. JC  *
*            :  03/10/87 Corrections to partially comply with latest  *
*               definitions of the language. JC                       *
*            :  04/23/87 tokens no longer output to intermediate file *
*               GetToken called directly by the Parser now.           *
***********************************************************************/

#include <scanner.h>
#include <stdio.h>
#include <errors.h>
#include <ctype.h>

extern   FILE *infile, *errorfile;                /* working files            */
/*****************************************************************************/

   int
GetToken(token)
   char *token;

/* Calls fgetc(infile) for the next char from the input file & builds  *
*  the token a char at a time.   Returns an internal integer value     *
*  representing the type of token found                                */
{
                                                  /* lookahead is a flag, line_num  */
static int lookahead = FALSE, line_num = 1;       /* is current line # of user prog */
static char ch;                                    /* ch holds last character from   */
                                                   /* input prog                     */
int    i, k;                                       /* i = index into token array;    */
                                                   /* k = temp holder for tokens     */
extern ErrorHandler();

   while(TRUE)
   {  i = 1;                                       /* initialize, token name --      */
                                                   /* will start at token[1]         */
      if (!lookahead)
         ch = fgetc(infile);
      lookahead = FALSE;

      while (isspace(ch))
      {  if(ch == '\n')                            /* process carrage returns        */
         {  ++line_num;
            return(EOLN_);
         }                                         /* end if '\n'                    */
         ch = fgetc(infile);
      }                                            /* end while white space          */

      if (ch == EOF )   return(EOF_);              /* reached end of the file        */

            switch(ch)
      {  case '+':   return(ADD_);
         case '*':   return(MULT_ );
         case '%':   return(IDIV_);
         case ';':   return(SEMI_);
         case '!':   return(SUBSCRIPT_);
         case ',':   return(COMMA_);
         case '(':   return(LTPAREN_);
         case ')':   return(RTPAREN_);
         case '~':   return(NEGLOG_);
         case ':':   return(COLON_);
         case '^':   return(CAT_);
         case '@':   return (STAR_);
         case '[':   return (LTBRAKET_);
         case ']':   return (RTBRAKET_);
         case '{':   return(LTSQUIG_);
         case '}':   return(RTSQUIG_);
         case '-':
            if ((ch = fgetc(infile)) == '>')
               return (RTARROW_);
            else if(ch == '-')                     /* process comment                */
            {  while((ch = fgetc(infile))
                     != '\n' && ch != EOF)
                  ;                                /* do nothing                     */
               if(ch == '\n')
               {  ++line_num;
                  return(EOLN_);
               }  else  return(EOF_);
            }                                      /* end else if comment            */
            else
               lookahead = TRUE;
            return(SUB_);

         case '<':
            if((ch = fgetc(infile)) == '=')
               return(LEQ_);
            else if(ch == '>')
               return(NEQ_);
            else lookahead = TRUE;
      return(ST_SEQUENCE_);

    case '>':
      if((ch = fgetc(infile)) == '=')
         return(GEQ_);
      else lookahead = TRUE;
      return(END_SEQUENCE_);

   case '=':
      if((ch = fgetc(infile)) == '=')
         return(EQUIV_);
      else lookahead = TRUE;
         return(EQ_);

   case '/':
      if((ch = fgetc(infile)) == '\\')
         return(ANDLOG_);
      else lookahead = TRUE;
      return(RDIV_);

   case '\\':
      if((ch = fgetc(infile)) == '/')
         return(ORLOG_);
      else
         lookahead = TRUE;
      ErrorHandler(line_num,ERR2,NULL);
      return(ORLOG_);                        /* figured that's what he wanted  */

   case '|':
      if((ch = fgetc(infile)) == '-')
         if((ch = fgetc(infile)) == '>')
            return(LINERTARROW_);
      lookahead = TRUE;
      ErrorHandler(line_num,ERR0,ch);        /* ch is either '|' or '-'        */
      return(LINERTARROW_);                  /* figured that's what he wanted  */

   case '$' :
      ch = fgetc(infile);
      if ((ch == 'R') || (ch == 'r'))
       return(REAL_);
      else if ((ch == 'N') || (ch == 'n'))
       return(NATURAL_);
      else if ((ch == 'Z') || (ch == 'z'))
       return(INTEGER_);
      else if ((ch == 'B') || (ch == 'b'))
       return(BOOLEAN_);
      else if (ch == '1')
       return(TRIVIAL_);
      else lookahead = TRUE;
      ErrorHandler(line_num,ERR3,NULL);
      return(INTEGER_);                      /* default return type            */
}                                            /* end switch                     */

if ( isalpha(ch))                            /* starts with a letter           */
{  do
   {  token[i++] = ch;
      ch = fgetc(infile);
   }   while (isalnum(ch) || ch =='_');      /* end do                         */
   token[i] = '\0';                          /* end the string                 */
   lookahead = TRUE;

                                             /* now check to see if it's a     */
                                             /* KEYWORD                        */
         if(( k = IsKeyWord(token +1)) >= 0)
            return(KW_ + k);                       /* Return Adjusted Keyword index  */
                                                   /* to calling routine             */

         return(IDENTIFIER_);
      }                                            /* end if alfa char               */
      if (isdigit(ch))
      {  if(ch == '0')
         {  while((ch=fgetc(infile)) == '0')
               ;                                   /* do nothing, eat zeros          */
            if(!isdigit(ch))
               token[i++] = '0';
         }                                         /* end if leading 0               */
         while(isdigit(ch))
         {  token[i++] = ch;
            ch = fgetc(infile);
         }                                         /* end while is a digit           */
         if (ch == '.')
         {  token[i++] = ch;
            ch = fgetc(infile);
            if (!isdigit(ch))
            {  token[i] = '\0';
               ErrorHandler(line_num,ERR4,
                            token+1);
               token[i + 1] = token[i];            /* fix for insertion into         */
               token[i] = 0;                       /* name table                     */
               return(CONSTANT_);
             }else                                 /* end if not a digit             */
               do
               {  token[i++] = ch;
                  ch = fgetc(infile);
               }  while(isdigit(ch));              /* end do                         */
         }                                         /* end if ch == '.'               */
         token[i] = '\0';                          /* end the string                 */
         lookahead = TRUE;
         return(CONSTANT_);
      }                                            /* end if isdigit                 */
      if (ch == '\'')                              /* process LITERALS               */
      {  token[i++] = ch;
         while(((ch=fgetc(infile))!=EOF) &&
                (ch!='\n') && (i< MAXLINE))
         {  token[i++] = ch;
            if (ch == '\'')
               if ((ch = fgetc(infile)) != '\'')
               {  lookahead = TRUE;
                  token[i] = '\0';
                  if (strlen (token) > 3)
                     return (LITERAL_);
                  return (EMPT_LIT_);
               }                                   /* end if != '                    */
         }                                         /* end while                      */
      token[i]='\0';                               /* end the string                 */
      ErrorHandler(line_num,ERR5,token+1);
      return(LITERAL_);                            /* figured that's what he wanted  */
      }                                            /* end if literal                 */

      ErrorHandler(line_num,ERR6,&ch);             /* Default - char not recognized  */

      continue;                                    /* let's try again                */
   }                                               /* end while (true)               */
}                                                  /* end GetToken()                 */
```

```
/**************************   Scanner Utilites   **********************/

   int
IsKeyWord(token)
   char *token;

/* Checks to see if the input token is a keyword in the language.      *
*  If it is, the function returns the numeric value of the keyword.    *
*  If it isn't, the function returns -1.  Performs binary search of    *
*  keyword array -                                                     *
* MUST KEEP THIS ARRAY IN ALPHABETICAL ORDER!!                         */

{
int       i;
register int   lo = 0, hi , mid;
                                                  /* list of PHI keywords - KEEP in */
                                                  /* alphabetical order!!           */
char    s[MAXLINE];static   char   *keywords[] =
         {"AND","BEGIN","ELSE","ELSIF","END","ENDIF","FILE","GREATER","IF",
         "IN","LESS","LET","NOTIN","READ","THEN","TYPE","WHERE","WRITE" };

   strcpy(s,token);
   for(i = 0; s[i] != '\0';i++)
      if('a' <= s[i] && s[i] <= 'z')              /* insure letters are upper case  */
         s[i] += 'A' - 'a';

   hi = (sizeof(keywords)/sizeof(char*));

   while(lo <= hi)
   {  mid = (lo + hi)/2;
      if((i = strcmp(s,keywords[mid]))<0)
         hi = --mid;
      else if(i>0)
         lo = ++mid;
      else
         return(mid);                             /* found a keyword                */
   }                                              /* end while                      */
   return(-1);                                    /* didn't find a keyword          */
}                                                 /* end IsKeyWord                  */
/************************************************************************************/
```

APPENDIX H

ROCK COMPILER — PARSER

```
/***********************************************************************
*                              PUBLIC DOMAIN SOFTWARE                  *
* Name      :   parser pt I                                            *
* File      :   parser1.c                                              *
* Authors   :   Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :   10/20/86                                               *
* Archived  :   12/11/86                                               *
* Modified  :   04/23/87 No longer set up to work with file of tokens. *
************************************************************************
* This file contains the following modules for the PHI parser:        *
*                                                                      *
*    BlockBody()    LetDefs()    Defs()     DefAnd()    QualExp()      *
*    AuxExp()       AuxDefs()    AuxAnd()   Formals()   Expression()   *
*                                                                      *
* Algorithm : The main module calls BlockBody() to start the parse    *
*              off.  BlockBody in turn calls LetDefs() first and then *
*              QualExp() looking for a valid program.  The remaining   *
*              modules in Pt's 1-3 are called by these when trying to *
*              validate a pargram.  The results from the parse are now *
*              kept in an abstract syntax tree for type checking and   *
*              code generation.  Various utility functions are used    *
*              to build the tree and  simplify parsing the grammer.    *
*                                                                      *
************************************************************************
*                                                                      *
* Modified   :  12/26/86 Flattened tree output changed to abstract     *
*               syntax tree form. JC                                   *
*            :  01/10/87 Corrections to comply with latest definitions *
*               of the language. JC                                    *
*            :  01/27/87 Error Recovery added and files combined. JC   *
*            :  03/20/87 Token buffer implemented for parser. JC       *
*            :  03/29/87 Changed manner errors are handled - required  *
*               for integration with back-end.                         *
*            :  04/23/87 No longer set up to work with file of tokens. *
*               GetToken is called directly thru FillBuff().    JC     *
***********************************************************************/

#include <stdio.h>
#include <parser.h>
                                                   /* global flags - aid in making   */
int     rtbrket = FALSE, argbind = FALSE;          /* PHI deterministic              */

int     line_no = 1;                               /* global var, current line no    */
                                                   /* of program                     */
                                                   /* tokenbuff holds tokens provided*/
                                                   /* by GetToken() - ptr is a ptr to*/
                                                   /* next token in tokenbuffer -    */
```

```
                                                  /* must use "long" because buffer */
                                                  /* holds addresses                */
                                                  /* use BUFSIZE + 1 in case have to*/
                                                  /* place address of *name at end  */
                                                  /* of buffer                      */

long   tokenbuff[BUFSIZE+1], *ptr = &tokenbuff[BUFSIZE];

FILE   *poutfile, *errorfile;                     /* working files                  */

/**********************************************************************************/

    nodal
Parser ()
{
NodeRec   *root = NULL;
extern    void p_close(), mov_cursor();           /* external asm functions         */

   num_errors = 0;                                /* init number jof errors         */

   errorfile = fopen("errors.phi","w");
   fprintf(errorfile,"%40s\n\n","ROCKY ERRORS");
   fclose(errorfile);                             /* rewrite file for clean start   */

#ifdef   DEBUG
   poutfile = fopen("parser.out","w");
#endif

   BlockBody(&root);                              /* look for a valid program       */

   if(!ByPass(EOF_))
   {  mov_cursor(20,0);                           /* set cursor on screen to        */
      printf("WARNING ...additional text found    /* found extra junk, tell user    */
              at completion of your program -
              line %d\n",line_no);
   }                                              /* end if not end of user's prog  */
#ifdef   DEBUG
   if (root != NULL)                              /* write parser's output          */
                                                  /* to data file                   */
      PostOrder(roct);                            /* case it's needed for debugging */
   fprintf(poutfile,"\n");                        /* need that carrage return       */
   fclose(poutfile);
#endif

   fclose(infile);

   p_close();

   if (num_errors > 0)
      return(NULL);
   else
      return(root);
}                                                 /* end main                       */
/**********************************************************************************/

   void
PostOrder(root)
   NodeRec    *root;

/*     Does a post order walk of the tree with (root) as its head.      */
/*     Just prints out the node name to the screen now                  */
{
```

```
static int i = 0;                                  /* used in pretty printing parser */
                                                   /* output file                    */

   if (root != NULL)
   {  PostOrder(root->lptr);
      PostOrder(root ->rptr);

      switch (root->name)
      {  case IDENTIFIER_   :
         case CONSTANT_     :
         case LITERAL_      :
         { fprintf(poutfile,"%d ",root ->name);
           fprintf(poutfile,"%ld    ",root->index);
           break;           }                      /* end ID,CONSTANT,LITERAL        */
         default            :
           fprintf(poutfile,"%d   ",root ->name);
         }                                         /* end switch                     */
   if (((++i % 7)==0))  fprintf(poutfile,"\n");
   }                                               /* end root != NULL               */
}                                                  /* end PostOrder()                */
/*************************************************************************************/

   int
BlockBody(root)                                    /* root is a ptr to tree/subtree  */
   NodeRec    **root;                              /* currently working with         */

/*                <BLOCKBODY> ::= <QUALEXP> | <LETDEFS>                    */
{
int     flag;

   if(((flag = LetDefs(root)) == TRUE))
      return(TRUE);
                                                   /* looking for LETDEFS;BLOCKBODY  */
    else if(flag != ERROR_)                        /* look for plain ol' QUALEXP     */
      flag = QualExp(root);

   return(flag);
}
/*************************************************************************************/

   int
LetDefs(root)                                      /* root is a ptr to tree/subtree  */
   NodeRec    **root;                              /* currently working with         */

/*                    <LETDEFS> ::= let <DEFS> ; <BLOCKBODY>                       */
{
#ifdef DEBUG
printf(" letdefs entered\n");   scanf("%*c");
#endif

   if(ByPass(KW_ + LET_))                          /* starts off with LET            */
   {  *root = CreateNode(LETDEF);                  /* start off the tree             */
      if(Defs(&((*root)->lptr)) != TRUE)           /* look for the definitions       */
         ErrorHandler(line_no,ERR_a,               /* report it, try & fix           */
                      (long)SEMI_);

      if(!ByPass(SEMI_))
         ErrorHandler(line_no,ERR_b,               /* report it & try to continue    */
                      (long)SEMI_);
      ByPass(SEMI_);
      if((BlockBody(&((*root)->rptr)) == TRUE))
         return(TRUE);                             /* found everything               */
      ErrorHandler(line_no,ERR_a,NULL);            /* report it, no fix              */
      return(ERROR_);                              /* started LETDEFS but couldn't   */
                                                   /* finish                         */
    }                                              /* end ByPass LET                 */
#ifdef DEBUG
 printf(" letdefs exited\n");   scanf("%*c");
#endif

    return(FALSE);                                 /* default - no LET at beginning  */
}                                                  /* end LetDefs                    */
/***********************************************************************************/

   int
Defs(root)                                         /* root is a ptr to tree/subtree  */
   NodeRec    **root;                              /* currently working with         */

/* <DEFS> ::= (<DATADEF>|<FUNDEF>|<KINDEF>|<TDEFID>|                                 */
/*             <TDEFFUN>)  <DEFAND>                                                  */
/*              Where "<DEFAND> " need not be present.                               */
{
NodeRec   *temp;
int       flag;
long      id_ptr;                                  /* address of data struct         */
                                                   /* holding identifier name        */
extern    long *ptr;

   if(id_ptr = ByPass(IDENTIFIER_))
   {   temp = CreateNode(IDENTIFIER_);             /* set up itsside of subtree      */
      temp ->index = id_ptr;

      if(ByPass(EQUIV_))                           /* looking for ID ==              */
      {  *root = CreateNode(DATADEF);              /* found '==' It's a DATADEF      */
         (*root)->lptr = temp;                     /* attach temp ptr to root        */
         if(QualExp(&((*root)->rptr)) != TRUE)     /* now need QualExp               */
                                                   /* note,try & fix                 */
            ErrorHandler(line_no,ERR_c,
                        (long)KW_+AND_);
       }                                           /* end ByPass EQUIV_              */
      else   if(ByPass(COLON_))                    /* looking for ID : TYPEEXP       */
      {  *root = CreateNode(KINDEF);               /* found : so it's a KINDEF       */
         (*root) ->lptr = temp;                    /* attach temp ptr to root        */
         if((TypeExp(&((*root)->rptr))!= TRUE))    /* now need TypeExp               */
                                                   /* note,try to fix                */
            ErrorHandler(line_no,ERR_d,
                        (long)KW_+AND_);
      }                                            /* end else if ByPass :           */
      else                                         /* not == or :, so must be        */
                                                   /* ID FORMALS                     */
      {  *root = CreateNode(FUNDEF);
         (*root) ->lptr = CreateNode(FUNID);       /* will look for ID FORMALS       */
         (*root) ->lptr->lptr = temp;              /* attach ID to FUNID             */
         if((Formals(&(*root)->lptr->rptr)
                         !=TRUE))                  /* need the FORMALS               */
            ErrorHandler(line_no,ERR_e,            /* note it,try & fix              */
                        (long)EQUIV_);
                                                   /* Look for '==',already created  */
         if(!ByPass(EQUIV_))                       /* FUNAUXDEF - Need QualExp on rt */
                                                   /* note,try & fix                 */
            ErrorHandler(line_no,ERR_f,
                        (long)KW_+AND_);
         else if((QualExp(&((*root)->rptr))
                          !=TRUE))
```

```
           ErrorHandler(line_no,ERR_c,            /* note it,try & fix              */
                        (long)KW_+AND_);
     }                                            /* end else not '==' or ':'       */

     goto CHECK;                                  /* found something so need to     */
                                                  /* check for more                 */
  }                                               /* end if ID                      */

  /******   didn't find ID, so look for FORMALS == QUALEXP   *******/

else if(((flag = Formals(root)) != FALSE))        /* found something                */
  {  if(flag==ERROR_)
        ErrorHandler(line_no,ERR_e,               /* note, try & fix                */
                    (long)EQUIV_);
     if(ByPass(EQUIV_))                           /* looking for ==                 */
     {  MakeNewRoot(root,DATADEF,LEFT);           /* found ==, so fix tree          */
        if((QualExp(&((*root)->rptr))!=TRUE))     /* need QualExp on rt             */
                                                  /* note,try & fix                 */
           ErrorHandler(line_no,ERR_c,
                        (long)KW_+AND_);
     }else                                        /* end if ByPass(EQUIV)           */
                                                  /* note,try to fix                */
        ErrorHandler(line_no,ERR_f,
                    (long)KW_+AND_);
     goto CHECK;                                  /* found somenthig so check for   */
                                                  /* more defs                      */
  }

  /*****   nothing so far  -  look for some sort of TYPEDEF   *****/

  else if(ByPass(KW_ + TYPE_))
  {   if(id_ptr = ByPass(IDENTIFIER_))            /* found TYPE, looking for ID     */
     { temp = CreateNode(IDENTIFIER_);            /* set up lt side of subtree      */
        temp ->index = id_ptr;

        if(ByPass(EQUIV_))                        /* found ID, looking for ==       */
        {  *root = CreateNode(TDEFID);            /* found == so it's a TDEFID      */
           (*root)->lptr = temp;
        }else                                     /* not '==' yet                   */
        {  if((Formals(root) != TRUE))            /* note it,try to fix             */
              ErrorHandler(line_no,ERR_e,
                          (long)EQUIV_);
           MakeNewRoot(root,FUNID,RIGHT);         /* found/fixed Formals,fix tree   */
           (*root) ->lptr = temp;                 /* attach ID to FUNID node        */
           if(!ByPass(EQUIV_))                    /* note it,try to fix             */
              ErrorHandler(line_no,ERR_f,
                          (long)KW_+AND_);
           MakeNewRoot(root,TDEFFUN,LEFT);        /* found/fixed '==',fix tree      */
        }                                         /* end else not '==' yet          */
        if((TypeExp(&((*root)->rptr))!= TRUE))    /* need TypeExp on rt             */
                                                  /* note,try & fix                 */
           ErrorHandler(line_no,ERR_d,
                        (long)KW_+AND_);
     }                                            /* end if ByPass ID               */
     else                                         /* no ID,note it,try to fix       */
        ErrorHandler(line_no,ERR_g,
                    (long)KW_+AND_);
    goto CHECK;                                   /* found sometthing so check for  */
                                                  /* more defs                      */
  }                                               /* end ByPass TYPE                */

return(flag);                                     /* default - none of the above    */
```

```
CHECK:                                          /* found something so need to    */
                                                /* check for more def's          */
   DefAnd(root);

   return(TRUE);                                /* any errors have been noted,   */
                                                /* so press on                   */
}                                               /* end Defs                      */

/************************************************************************/

 void
DefAnd(root)                                    /* root is a ptr to tree/subtree */
   NodeRec   **root;                            /* currently working with        */

/*                   <DEFAND> ::= and <DEFS>                              */
/*              Where " and <DEFS> " need not be present.                 */
/*      Note:  This function assumes root is not NULL upon entry          */
{
   if(ByPass(KW_ + AND_))
   {  MakeNewRoot(root,DEFAND,LEFT);            /* found "and" so fix tree       */
      if(Defs(&(*root)->rptr) != TRUE)
         ErrorHandler(line_no,ERR_h,
                     (long)SEMI_);              /* note it, try to fix           */
   }                                            /* end ByPass AND                */
}                                               /* end DefAnd                    */
/************************************************************************/

   int
QualExp(root)                                   /* root is a ptr to tree/subtree */
   NodeRec   **root;                            /* currently working with        */

/*          <QUALEXP> ::= <EXPRESSION> where <AUXEXP>                     */
/*              Where "where <AUXEXP>" need not be present.               */
{
int    flag;
#ifdef DEBUG
printf(" qualexp entered\n");   scanf("%*c");
#endif

   if(((flag = Expression(root))== ERROR_))     /* errors already reported,      */
      EatEm(KW_+END_);                          /* attempt to press on           */
   if(ByPass(KW_+ WHERE_))                      /* looking for where expression  */
   {  MakeNewRoot(root,(KW_+WHERE_),RIGHT);     /* found one,fix tree            */
      AuxExp(&((*root)->lptr));                 /* need AuxExp following WHERE   */
   }                                            /* end byPass WHERE              */

#ifdef DEBUG
 printf(" qualexp exited   %d\n",flag);
 scanf("%*c");
#endif

   return(flag);                                /* default - just return flag    */
}                                               /* end Qualexp()                 */
/************************************************************************/

   int
AuxExp(root)                                    /* root is a ptr to tree/subtree */
   NodeRec   **root;                            /* currently working with        */

/*         <AUXEXP> ::= <AUXDEFS> (where <AUXEXP>)*                       */
```

```
{
int    flag;

   if(((flag = AuxDefs(root))!= TRUE))             /* need at least one AUXDEF       */
      ErrorHandler(line_no,ERR_i,                  /* note, try & fix                */
                  (long)KW_+WHERE_);

   if(ByPass(KW_+ WHERE_))                         /* looking for multiple WHERE's   */
   {  MakeNewRoot(root,(KW_ + WHERE_),RIGHT);      /* found one,fix tree             */
      AuxExp(&((*root)->lptr));                    /* need AuxExp following WHERE    */
   }                                               /* end ByPass(WHERE)              */

   return(flag);                                   /* default - return result of     */
                                                   /* first AuxDefs                  */
}                                                  /* end AuxExp                     */
/**************************************************************************************/

   int
AuxDefs(root)                                      /* root is a ptr to tree/subtree  */
   NodeRec   **root;                               /* currently working with         */

/*          <AUXDEFS> ::= (<DATAAUAXDEF> | <FUNAUXDEF>) <AUXAND>           */
/*                   Where "<AUXAND> " need not be present.                */
{
NodeRec   *temp;
int       flag;
long      ptr;                                     /* address of data struct holding */
                                                   /* identifier name                */


   if((ptr = ByPass(IDENTIFIER_)))
   {   temp = CreateNode(IDENTIFIER_);             /* set up its side of subtree     */
      temp ->index = ptr;

      if(ByPass(EQUIV_))                           /* looking for ID ==              */
      {  *root = CreateNode(DATAAUXDEF);           /* found '==' It's a DATAAUXDEF   */
         (*root)->lptr = temp;                     /* attach temp ptr to root        */
         if(Expression(&((*root)->rptr))!= TRUE)   /* now need Exp                   */
            ErrorHandler(line_no,ERR_c,            /* noteit, try & fix              */
                         (long)KW_+WHERE_);
      }                                            /* end ByPass EQUIV_              */
      else                                         /* not '==' so must be ID FORMALS */
      {  *root = CreateNode(FUNAUXDEF);
         (*root) ->lptr = CreateNode(FUNID);       /* will look for ID FORMALS       */
         (*root) ->lptr->lptr = temp;              /* attach ID to FUNID             */
         if((Formals(&(*root)->lptr->rptr)         /* need the FORMALS               */
             != TRUE))
            ErrorHandler(line_no,ERR_e,            /* note, try to fix               */
                         (long)EQUIV_);
                                                   /* Looking for '==',already       */
                                                   /* created FUNAUXDEF -            */
         if(!ByPass(EQUIV_))                       /* need QualExp on rt             */
            ErrorHandler(line_no,ERR_f,            /* note the errors, try & fix     */
                         (long)KW_+WHERE_);
         else
            if((Expression(&((*root)->rptr))
                != TRUE))
               ErrorHandler(line_no,ERR_c,
                            (long)KW_+WHERE_);
      }                                            /* end else not '=='              */
      goto CHECK;                                  /* found something so need to     */
                                                   /* check for more                 */
   }                                               /*   end if ID                    */
```

```
  /******    didn't find ID, so look for FORMALS == EXP    ***********/

   if(((flag = Formals(root)) != FALSE))          /* found something                */
   {  if(flag==ERROR_)
         ErrorHandler(line_no,ERR_e,              /* note, try & fix                */
                     (long)EQUIV_);
      if(ByPass(EQUIV_))                          /* looking for ==                 */
      {  MakeNewRoot(root,DATAAUXDEF,LEFT);       /* found ==, so fix tree          */
         if((Expression(&((*root)->rptr))         /* need Exp on rt                 */
            != TRUE))
            ErrorHandler(line_no,ERR_c,           /* note,try fix                   */
                        (long)KW_+WHERE_);
      }else
         ErrorHandler(line_no,ERR_f,
                     (long)KW_+WHERE_);

      goto CHECK;                                 /* found somenthig so check for   */
                                                  /* more auxdefs                   */
   }
   return(flag);                                  /* default - none of the above    */

CHECK:                                            /* found something so need to     */
                                                  /* check for more def's           */
   AuxAnd(root);

   return(TRUE);                                  /* any errors have been noted,    */
                                                  /* so press on                    */
}                                                 /* end AuxDefs                    */
/************************************************************************************/

   void
AuxAnd(root)                                      /* root is a ptr to tree/subtree  */
   NodeRec    **root;                             /* currently working with         */

/*                     <AUXAND> ::= and <AUXDEFS>                                   */
/*          Where "and <AUXDEFS>" need not be present.                              */
/*          Note:  This function assumes root is not NULL upon entry                */
{
   if(ByPass(KW_+AND_))
   {  MakeNewRoot(root,AUXAND,LEFT);              /* found "and" so fix tree        */
      if((AuxDefs(&(*root)->rptr) != TRUE))
         ErrorHandler(line_no,ERR_h,              /* note it, try & fix             */
                     (long)KW_+WHERE_);
   }                                              /* end ByPass AND                 */
}                                                 /*   end AuxAnd                   */
/************************************************************************************/
   int
Formals(root)                                     /* root is a ptr to tree/subtree  */
   NodeRec    **root;                             /* currently working with         */

     /*              <FORMALS> ::= <ID> | '(' <FORMALS> ',' ')'       */
{
NodeRec    *temp, *workingroot;                   /* temp ptrs to nodes in tree     */
                                                  /* workingptr marches down the    */
                                                  /* rt side of the subtree         */
long       ptr;

   if ((ptr = ByPass(IDENTIFIER_)))               /* checking for just an ID        */
   {  *root = CreateNode(IDENTIFIER_);
      (*root) ->index = ptr;
      return(TRUE);
```

```
{  *root = CreateNode(IDENTIFIER_);
   (*root) ->index = ptr;
                                                 /* now look for ID |-> ACTUAL     */
   if(ByPass(LINERTARROW_))                      /* Note: "ID |-> ACTUAL" is a     */
   {                                             /* <DENOTATION>                   */
      MakeNewRoot(root,LINERTARROW_,LEFT);       /* found one so fix tree          */
      if(Actual(&((*root)->rptr)) == TRUE)       /* look for trail ACTUAL          */
         return(TRUE);
      else                                       /* note it, no fix                */
      {  ErrorHandler(line_no,ERR8,
                      (long)LINERTARROW_);
         return(ERROR_);
      }                                          /* end else not Actual()          */
   }                                             /* end if LINERTARROW             */
   return(TRUE);
}                                                /* end if ID                      */

if ( ByPass(KW_ + FILE_))                        /* found keyword FILE             */
{  *root = CreateNode(KW_ + FILE_);
   if ((ptr = ByPass(LITERAL_)))
   {  temp = CreateNode(LITERAL_);               /* attach following LITERAL       */
      temp ->index = ptr;
      (*root) ->rptr = temp;
      return(TRUE);
   }                                             /* end if LITERAL_                */
   else                                          /* note it, no fix                */
   {  ErrorHandler(line_no,ERR_1,NULL);
      return(ERROR_);
   }
}                                                /* end if FILE_                   */

if ((flag = Conditional(root)) != FALSE)
   return(flag);
if ((flag = Block(root)) != FALSE)
   return(flag);

                                                 /* Phi is nondeterministic must   */
                                                 /* first check for compounds then */
                                                 /* if |-> follows must see if the */
                                                 /* compound was actually a formals*/
                                                 /* list  NOTE:  Order may NOT be  */
                                                 /* changed!!                      */
if ((flag = Compound(root)) == TRUE)
if(!ByPass(LINERTARROW_))    return(TRUE);
else                                             /* had "|->"  now need to see if  */
                                                 /* had Formals                    */
{  temp = *root;                                 /* set var to be passed by value  */
                                                 /* to IsFormals                   */
   if(!IsFormal(temp))                           /* just report it and press on    */
      ErrorHandler(line_no,ERR_o,NULL);
   (*root)->name = FORMAL;
   MakeNewRoot(root,LINERTARROW_,LEFT);          /* found one so fix tree          */
   if(Actual(&((*root)->rptr)) == TRUE)          /* look for trail ACTUAL          */
      return(TRUE);
   else                                          /* note it, no fix                */
   {  ErrorHandler(line_no,ERR8,
                   (long)LINERTARROW_);
      return(ERROR_);
   }
}                                                /* end else ByPass LINERTARROW    */
else if(flag == ERROR_)
   return(ERROR_);
   if ((flag = Denotation(root)) != FALSE)
      return(flag);

   if ((flag = ArgBinding(root)) != FALSE)
      return(flag);

   return(FALSE);                                 /* Default, tried everything else */
}                                                 /* end Actual()                   */
/*************************************************************************************/
```

```
/***********************************************************************
*                              PUBLIC DOMAIN SOFTWARE                  *
* Name      :   parser pt 3                                            *
* File    . :   parser3.c                                              *
* Authors   :   Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :   10/20/86                                               *
* Archived  :   12/11/86                                               *
* Modified  :   01/27/87 - Error Recovery added.   JC                  *
************************************************************************
* This file contains the following modules for the PHI parser:         *
*        Conditional()   Arm()            Block()      Compound()      *
*        Elements()      Denotation()     ArgBind()    Op()            *
*        TypeExp()       TypeDom()        TypeTerm()   TypeFac()       *
*        TypePrimary()   PrimType()                                    *
*                                                                      *
* Algorithm :  See parser part 1                                       *
*                                                                      *
************************************************************************
* Modified   :  12/26/86 Flattened tree output changed to abstract     *
*               syntax tree form. JC                                   *
*            :  01/10/87 Corrections to comply with latest definitions *
*               of the language. JC                                    *
*            :  01/27/87 Error Recovery added and files combined. JC   *
***********************************************************************/

#include <stdio.h>
#include <parser.h>

extern int   rtbrket;                              /* global flag - aids in          */
                                                   /* making PHI deterministic       */
extern int   line_no;                              /* global var, current line       */
                                                   /* number of program              */
/*************************************************************************************/

   int
Conditional(root)                                  /* root is a ptr to tree/subtree  */
   NodeRec   **root;                               /* currently working with         */

/*  <CONDITIONAL> ::= if <ARM> (elsif<ARM>)* (else<EXPRESSION)1 endif */

{                                                  /* ptrs to temp nodes in the tree */
NodeRec   *temp = NULL, *subroot, *workingptr;

   if(ByPass(KW_ + IF_))
   {  if(Arm(&temp) != TRUE)
         ErrorHandler(line_no,ERR_m,(long)IF_);    /* note it, try to fix            */
      *root = CreateNode(KW_ + IF_);               /* set up root for return         */
      (*root) ->lptr = temp;                       /* attach THEN exp to root        */
      workingptr = *root;                          /* move working ptr               */

      while(ByPass(KW_ + ELSIF_))
      {  subroot = CreateNode(KW_ + ELSIF_);
         workingptr ->rptr = subroot;              /* attach ELSIF to tree           */
         if(Arm(&temp) != TRUE)
            ErrorHandler(line_no,ERR_m,            /* note it,try & fix              */
                        (long)ELSIF_);
         subroot ->lptr = temp;                    /* attach THEN exp to ELSIF       */
         workingptr = workingptr ->rptr;           /* move wrking ptr down subtree   */
      }                                            /* end while ELSIF                */
      if(ByPass(KW_ + ELSE_))
      {  if(Expression(&temp) != TRUE)
            ErrorHandler(line_no,ERR_m,             /* note it, try & fix          */
                         (long)ELSE_);
         subroot = CreateNode(KW_ + ELSE_);
         workingptr ->rptr = subroot;               /* attach ELSE to tree          */
         subroot ->lptr = temp;                     /* attach EXPRESSION to ELSE    */
         workingptr = workingptr ->rptr;            /* move wrking ptr down subtree */
      }                                             /* end ELSE                     */
      if(ByPass(KW_ + ENDIF_))
      {  temp = CreateNode(KW_ + ENDIF_);
         workingptr ->rptr = temp;
      }else                                         /* note it and try to fix -     */
       { ErrorHandler(line_no,ERR_n,NULL);          /* will return TRUE regardless  */
         if(IBall(ENDIF_,1) || IBall(ENDIF_,2))    /* look 2 ahead for the END     */
            EatEm(ENDIF_);
      }                                             /* end else not ByPass ENDIF_   */
      return(TRUE);                                 /* saw an IF, any errors they   */
                                                    /* were already reported        */
   }                                                /* end if IF                    */
   return(FALSE);                                   /* didn't see an IF             */
}                                                   /* end Conditional()            */
/**********************************************************************************/

   int
Arm(root)                                           /* root is a ptr to tree/subtree  */
   NodeRec    **root;                               /* currently working with         */

/*            <ARM> ::=   <EXPRESSION>then<EXPRESSION>                         */
{
int   flag;
NodeRec *temp = NULL;                               /* temp ptr to a node in tree     */

   if((flag = Expression(&temp)) != TRUE)           /* if an error try to recover by  */
      EatEm(KW_+THEN_);                             /* look for THEN,ELSE,ELSIF,ENDIF */

   if (ByPass(KW_ + THEN_))
   {  *root = CreateNode(KW_ + THEN_);
      (*root) -> lptr = temp;
      if (Expression(&temp) == TRUE)
         (*root) -> rptr = temp;
      else                                          /* report it and try to press on  */
      ErrorHandler(line_no,ERR_m,
                   (long)THEN_);
   }                                                /* end begin if THEN              */
   else                                             /* report it and try to press on  */
      ErrorHandler(line_no,ERR_f,
                   (long)KW_+THEN_);
   return(flag);
}                                                   /* end Arm()                      */
/**********************************************************************************/

   int
Block(root)                                         /* root is a ptr to tree/subtree  */
   NodeRec    **root;                               /* currently working with         */

/*              <BLOCK>   ::=   begin <BLOCKBODY> end                          */
{

   if (ByPass(KW_ + BEGIN_))
   {  *root = CreateNode(KW_ + BEGIN_);            /* sets root for return errors    */
                                                    /* have already been reported     */
      if (BlockBody(&((*root)->lptr))!= TRUE)       /* look for BLOCKBODY             */
```

```
         ErrorHandler(line_no,ERR_m,             /* note it, try & fix             */
                     (long)BEGIN_);
      if(ByPass(KW_ + END_))
      {  (*root) ->rptr = CreateNode(KW_ + END_);
         return(TRUE);
      }                                            /* end bypass END                 */
      ErrorHandler(line_no,ERR_f,                /* note it, no fix                */
                  (long)KW_+END_);
      return(ERROR_);
   }                                               /* end ByPass BEGIN               */
   else    return(FALSE);
}                                                  /* end BLOCK                      */
/*************************************************************************************/

   int
Compound(root)                                     /* root is a ptr to tree/subtree  */
   NodeRec   **root;                               /* currently working with         */

/* <COMPOUND> ::= '('<ELEMENTS>')'|'{'<ELEMENTS>'}'|'<'<ELEMENTS>'>    */
                 /*          where <ELEMENTS> may be empty  */
{
   if(ByPass(LTPAREN_))
   {  Elements(root);                              /* only look for elemt's because  */
                                                   /* errors reported via QualExp    */
      if (!ByPass(RTPAREN_))                       /* note it, no fix                */
         ErrorHandler(line_no,ERR_f,
                     (long)RTPAREN_);

      if(*root == NULL)                            /* now check for empty compounds/ */
         *root = CreateNode(EMPTYCOMPOUND);        /* compounds w/ multiple elements */
      else if((*root)->name == COMMA_)
         (*root)->name = ELLIST;
      return(TRUE);
   }                                               /* end if LTPAREN)                */

   if(ByPass(LTSQUIG_))                            /* only look for 'em,             */
   {  Elements(root);                              /* errors reported via QualExp    */
      if (!ByPass(RTSQUIG_))
         ErrorHandler(line_no,ERR_f,             /* note it, no fix                */
                     (long)RTSQUIG_);

      if(*root == NULL)                            /* check for empty compounds and  */
         *root = CreateNode(EMPTYCOMPOUND);        /* compounds w/ multiple elements */
      else if((*root)->name == COMMA_)
         (*root)->name = ELLIST;
      return(TRUE);
   }                                               /* end if LTSQUIG)                */

   if(ByPass(ST_SEQUENCE_))                        /* only look for 'em,             */
   {  Elements(root);                              /* errors reported via QualExp    */
      if(!ByPass(END_SEQUENCE_))
         ErrorHandler(line_no,ERR_f,             /* note it & no fix               */
                     (long)END_SEQUENCE_);

      if(*root == NULL)                            /* now check for empty sequences/ */
         *root = CreateNode(EMPTYSEQUENCE);        /* sequences w/ multiple elements */
      else   MakeNewRoot(root,SEQUENCE,RIGHT);
      return(TRUE);
   }                                               /* end ByPass ST_SEQUENCE_        */
   return(FALSE);                                  /* none of the above              */
}                                                  /* end Compound()                 */
/*************************************************************************************/
```

```
   int
Elements(root)                                   /* root is a ptr to tree/subtree  */
   NodeRec   **root;                             /* currently working with         */

/*             <ELEMENTS> ::= <QUALEXP> (,<QUALEXP>)*                              */
{
int      flag;

   if((flag = QualExp(root)) == ERROR_)
      EatEm(COMMA_);                             /* errors already reported        */
   while(ByPass(COMMA_))                         /* recursively look for next      */
                                                 /* qualexp                        */
   {  MakeNewRoot(root,COMMA_,LEFT);             /* found a COMMA so fix tree      */
      if (Elements(&((*root)->rptr)) != TRUE)
         ErrorHandler(line_no,ERR_p,
                      (long)COMMA_);             /* note it, try & fix             */
      if((*root)->rptr->name != COMMA_)          /* fix tree so all QualExp's      */
        MakeNewRoot(&((*root)->rptr),
                         COMMA_,LEFT);           /* hang to the LEFT               */
   }
                                                 /* end while ByPass(COMMA_)       */
   return(flag);
}                                                /* end Elements()                 */
/***************************************************************************/

   int
Denotation(root)                                 /* root is a ptr to tree/subtree  */
   NodeRec   **root;                             /* currently working with         */

/*  <DENOTATION> ::=   <LITERAL> | <CONSTANT> | <FORMALS>|-> <ACTUAL>   *
*   where LITERAL is quoted(') string of zero or more chars and         *
*   where CONSTANT is an integer or decimal number                      *
*   NOTE:  <FORMALS> |-> <ACTUAL>   was already checked by Actual()    */
{
long   ptr;

    if(ptr = ByPass(LITERAL_))
    { *root = CreateNode(LITERAL_);
      (*root) ->index = ptr;
      return(TRUE);
   }                                             /* end a LITERAL                  */
   if (ByPass(EMPT_LIT_))
   {  *root = CreateNode(LITERAL_);
      (*root) ->index = NULL;
      return(TRUE);
   }                                             /* end a LITERAL                  */
   if(ptr =ByPass(CONSTANT_))
   {  *root = CreateNode(CONSTANT_);
      (*root) ->index = ptr;
      return(TRUE);
   }                                             /* end a CONSTANT                 */

   return(FALSE);                                /* default, none of the above     */

}                                                /* end Denotation()               */
/***************************************************************************/

   int
ArgBinding(root)                                 /* root is a ptr to tree/subtree  */
   NodeRec   **root;                             /* currently working with         */
```

```
/*   <ARGBINDING> ::= '[' (<OP><QUALEXP> | <QUALEXP><OP> | <OP>) ']'                */

{
int      specialcase;
NodeRec  *temp = NULL;                            /* temp ptr to node in tree       */
extern   int   argbind;                           /* global flag needed to make     */
                                                  /* PHI deterministic              */
   if(ByPass(LTBRAKET_))                          /* set global flag, needed to     */
   {  argbind = TRUE;                             /* PHI deterministic.             */
      specialcase = (IBall(ADD_,1) || IBall(SUB_,1));

#ifdef   DEBUG
printf("special case = %d argbind = %d\n",specialcase,argbind);
#endif

      if (Op(root))                               /* begin Op comes first           */
      {  if (ByPass(RTBRAKET_))                   /* looking for [Op]               */
         {  argbind = FALSE;                      /* reset global flag              */
            MakeNewRoot(root,ARGBINDOP,LEFT);
            return(TRUE);                         /* had [ <Op> ]                   */
         }                                        /* end if ByPass RTBRAKET_        */
         MakeNewRoot(root,ARGLEADOP,LEFT);        /* don't have just an Op          */
         if(IBall(ADD_,1) || IBall(SUB_,1))       /* might be +/-  +/- QualExp      */
            specialcase = FALSE;                  /* and don't want to accept       */
                                                  /* +/-  +/- QualExp Op later on   */
         if((QualExp(&(*root)->rptr))==TRUE)      /* two cases where QualExp could  */
         {                                        /* be TRUE   ---   <Op><QualExp>  */
            if(ByPass(RTBRAKET_))                 /* or +|-<QualExp><Op>            */
            {  argbind = FALSE;   return(TRUE); } /* reset global flag              */
            else                                  /* could be +/- PRIMARY           */
            if(specialcase && Op(&temp)
                           && ByPass(RTBRAKET_))
            {  ((*root)->lptr)->rptr=(*root)->rptr;
               (*root) ->rptr = temp;             /* now fix the tree               */
               (((*root)->lptr)->name == ADD_) ?
               (((*root)->lptr)->name=POS_) :
               (((*root)->lptr) ->name = NEG_);
               (*root)-> name = ARGTRAILOP;       /* <Op> came last as a ","        */
               argbind = FALSE;                   /* reset globalflag               */
               return(TRUE);
            }                                     /* end else specialcase && Op()   */
                                                  /* && RTBRAKET_                   */
         }                                        /* end 2 cases where QualExp TRUE */
         argbind = FALSE;                         /* reset globalflag               */
         ErrorHandler(line_no,ERR_q,NULL);        /* report it, no fix              */
         return(ERROR_);
      }                                           /* end Op comes first             */
      if ((QualExp(root)) != FALSE)               /* found something                */
      {  MakeNewRoot(root,ARGTRAILOP,LEFT);
         argbind = FALSE;                         /* reset global flag &            */
         if(Op(&(*root)->rptr)
            && ByPass(RTBRAKET_))                 /* see if can continue            */
            return(TRUE);
         ErrorHandler(line_no,ERR_q,NULL);        /* report it, no fix              */
         return(ERROR_);
      }                                           /* end if QualExp comes first     */
   }                                              /* end if ByPass LTBRAKET         */
   return(FALSE);                                 /* default, none of the above     */
}                                                 /* end ArgBinding()               */
/************************************************************************************/

   int
```

```
Op(root)                                    /* root is a ptr to tree/subtree  */
   NodeRec    **root;                       /* currently working with         */

/*          <OP> ::= , | ! | <RELATOR> | <ADDOP> | <MULOP>                    */
{
int    flag;

   if(flag = ByPass(COMMA_))
      *root = CreateNode(COMMA_);

   else if(flag = ByPass(SUBSCRIPT_))
      *root = CreateNode(SUBSCRIPT_);

   else if(flag = Relator())
      *root = CreateNode(flag);

   else if(flag = AddOp())
      *root = CreateNode(flag);

   else if(flag = MulOp())
      *root = CreateNode(flag);

   return(flag);
}                                           /* end Op                         */
/*************************************************************************/

   int
TypeExp(root)                               /* root is a ptr to tree/subtree  */
   NodeRec    **root;                       /* currently working with         */

/*         <TYPEEXP> ::=    <TYPEDOM> ( -> <TYPEEXP>   )*                     */
{
NodeRec    *newroot;                        /* temp ptr to nodes in the tree  */
int        flag;

   if((flag = TypeDom(root)) == TRUE)
   if (ByPass(RTARROW_))                    /* will recursively search for    */
   {  newroot = CreateNode(RTARROW_);       /* more TYPEEXP's                 */
      newroot ->lptr = *root;               /* fix root for return            */
      *root = newroot;
      if(TypeExp(&((*root)->rptr)) != TRUE)
      {   ErrorHandler(line_no,ERR9,(long)RTARROW_);
          return(ERROR_);
      }
   }                                        /* end recursive search           */
   return(flag);
}                                           /* end TypeExp                    */
/*************************************************************************/

   int
TypeDom(root)                               /* root is a ptr to tree/subtree  */
   NodeRec    **root;                       /* currently working with         */

/*               <TYPEDOM> ::=   <TYPETERM>(+ <TYPEDOM>)*                     */
{
NodeRec   *newroot;                         /* temp ptr to nodes in the tree  */
int       flag;

   if((flag = TypeTerm(root)) == TRUE)
   if (ByPass(ADD_))                        /* will recursively search for    */
   {  newroot = CreateNode(TYPEPLUS);       /* more TYPEDOM's                 */
      newroot ->lptr = *root;               /* fix root for return            */
      *root = newroot;
      if(TypeDom(&((*root)->rptr)) != TRUE)
      {  ErrorHandler(line_no,ERR9,(long)ADD_);
         return(ERROR_);
      }
   }                                           /* end recursive search         */
   return(flag);
}                                              /* end TypeDom()                */
/*************************************************************************/

   int
TypeTerm(root)                                 /* root is a ptr to tree/subtree */
   NodeRec    **root;                          /* currently working with       */

/*              <TYPETERM> ::=   <TYPEFAC>('*' <TYPETERM>)*                    */
{
NodeRec   *newroot;                            /* temp ptr to nodes in the tree */
int       flag;

   if((flag = TypeFac(root)) == TRUE)
   if (ByPass(MULT_))                          /* will recursively search for  */
   {  newroot = CreateNode(TYPETIMES);         /* more TYPETERMS's             */
      newroot ->lptr = *root;                  /* fix root for return          */
      *root = newroot;
      if(TypeTerm(&((*root)->rptr)) != TRUE)
      {   ErrorHandler(line_no,ERR9,
                       (long)MULT_);
          return(ERROR_);
      }
    }                                          /* end recursive search         */
   return(flag);
}                                              /* end TypeTerm()               */
/*************************************************************************/

   int
TypeFac(root)                                  /* root is a ptr to tree/subtree */
   NodeRec    **root;                          /* currently working with       */

/*         <TYPEFAC> ::=   <TYPEPRIMARY>@ | <TYPEPRIMARY> |                    */
/*         <ID> '<<' <TYPEEXP> (,<TYPEEXP>)* '>>' <ACTUAL>                     */
/*               Where <<TYPEEXP(,TYPEEXP,...)>> and/or <ACTUAL>               */
/*               need not be present                                           */
{
NodeRec   *newroot;                            /* temp ptr to nodes in the tree */
int       flag;
long      ptr;

   if(ptr = ByPass(IDENTIFIER_))
   {  *root = CreateNode(IDENTIFIER_);
      (*root) ->index = ptr;

      if(ByPass(ST_SEQUENCE_) && ByPass(ST_SEQUENCE_))
      {  ErrorHandler(line_no,ERR_r,NULL);
         return(ERROR_);
      }                                        /* end bypass <<                */
      goto CHECK;
   }                                           /* end if ID                    */

   if((flag = TypePrimary(root)) == TRUE)
      goto CHECK;
   return(flag);                               /* return either ERROR or FALSE */
```

```
CHECK: if(ByPass(STAR_))
       { newroot = CreateNode(STAR_);
         newroot ->lptr = (*root);
         *root = newroot;
       }                                       /* end if STAR                 */

   return(TRUE);                               /* made it this far, all OK    */
}                                              /* end TypeFac()               */
/**********************************************************************/

   int
TypePrimary(root)                              /* root is a ptr to tree/subtree  */
   NodeRec    **root;                          /* currently working with      */

/*            <TYPEPRIMARY> ::= <PRIMTYPE> | '(' <TYPEEXP> ')'            */
/*            NOTE:  ID already checked in TYPEFAC()                      */
{
   if(ByPass(LTPAREN_))
   {  if(TypeExp(root) != TRUE)
         ErrorHandler(line_no,ERR9,            /* note it, no fix             */
                 (long)LTPAREN_);

      if(ByPass(RTPAREN_))
         return(TRUE);
      else
      {  ErrorHandler(line_no,ERR_f,
                 (long)RTPAREN_);
         return(ERROR_);
      }
   }                                           /* end ByPass '('              */

   if(PrimType(root))
      return(TRUE);

   return(FALSE);                              /* default                     */
}                                              /* end TypePrimary()           */
/**********************************************************************/

   int
PrimType(root)                                 /* root is a ptr to tree/subtree  */
   NodeRec    **root;                          /* currently working with      */

/* <PRIMTYPE> ::= real | integer | natural | boolean | trivial | type */

{
   if(ByPass(REAL_))
   {  *root = CreateNode(REAL_);
      return(TRUE);
   }                                           /* end if REAL                 */

   if(ByPass(INTEGER_))
   {  *root = CreateNode(INTEGER_);
      return(TRUE);
   }                                           /* end if INTEGER              */

   if(ByPass(NATURAL_))
   {  *root = CreateNode(NATURAL_);
      return(TRUE);
   }                                           /* end if NATURAL              */

   if(ByPass(BOOLEAN_))
   {  *root = CreateNode(BOOLEAN_);
```

```
      return(TRUE);
   }                                                 /* end if BOOLEAN              */

   if(ByPass(TRIVIAL_))
   {  *root = CreateNode(TRIVIAL_);
      return(TRUE);
   }                                                 /* end if TRIVIAL              */

   if(ByPass(KW_ + TYPE_))
   {  *root = CreateNode(KW_ + TYPE_);
      return(TRUE);
   }                                                 /* end if TYPE                 */

   return(FALSE);                                    /* default - none of the above */
}                                                    /* end PrimType()              */
/**************************************************************************/
```

```
/**********************************************************************
*                          PUBLIC DOMAIN SOFTWARE                     *
* Name      :   Parser Utilities                                      *
* File      :   parsr_util.c                                          *
* Authors   :   Maj E.J. COLE / Capt J.E. CONNELL                     *
* Started   :   01/26/87                                              *
* Archived  :   03/03/87                                              *
* Modified  :   04/23/87  FillBuffer() now calls GetToken() direct.   *
***********************************************************************
* This file contains the utility modules for the parser:              *
*           CreateNode()       MakeNewRoot()      ByPass()            *
*           FillBuff()         IsFormal()         IBall()             *
*           NodeName()         EnterName()        FindName()          *
*                                                                     *
***********************************************************************
* Modified   :  03/20/87 - Buffer Handling routines added - JC        *
*               04/23/87 - FillBufer() calls GetToken() direct vice   *
*                          working with intermediate file of tokens.  *
*                          EnterName() and FindName() added to place  *
*                          IDs, LITERALS, and CONSTANTS into the name *
*                          table.    JC                               *
*                                                                     *
**********************************************************************/

#include <stdio.h>
#include <parser.h>

extern   int   line_no;                        /* global var, holds line no      */
                                               /* of source prog                 */
extern   FILE  *pinfile;                       /* global working file            */

                                               /* Init token[0] to value other   */
                                               /* than NULL.  Token[0] holds the */
char      token[MAXLINE]="x";                  /* length of the string.          */
NameRec   *nametable[TABLESIZE+ 1],            /* add 1 because [0] is unusable  */
          *EnterName();


/**********************************************************************/
                /*              UTILITIES                 */

   NodeRec *
CreateNode(op)
   NodeType op;                                /* operator type of node          */

/* Creates a tree node and returns the pointer (temp) to this node.   */
/* Accepts node type (op), an integer, and inserts it into the node.  */
{
NodeRec   *temp;

   temp = CALLOC(1,NodeRec);                   /* create a node                  */
   temp -> name = op;
   temp -> ln = line_no;
   temp -> lptr = (temp -> rptr) = NULL;
   return(temp);
}                                              /*   end CreateNode()             */
/**********************************************************************/

   void
MakeNewRoot(root,type,side)
```

```
   NodeRec   **root;                            /* old root of subtree -         */
                                                /* will turn into new root       */
   int       type, side;                        /* (type) is type of new root    */
                                                /* (side) is side to att old root */

/* Creates a new working root for subtree.                                      */
/* Old root is attached to lt/rt based on value of (side)                        */
{
NodeRec   *newroot;

   newroot = CreateNode(type);
   (side == LEFT) ?
   (newroot ->lptr = *root) : (newroot ->rptr = *root);
   *root = newroot;
}                                               /* end MakeNewRoot               */
/******************************************************************************/

   void
FillBuff(start)
  long  *start;                                 /* which slot in the buffer      */
                                                /* array to start the filling    */
/* Requires the buffer array and buffer ptr to be previously defined. */
/* Fills the buffer with tokens by calling GetToken().  Buffer filled */
/* until 1) end of user prog reached or 2) end of the array reached   */
/* If the token is a literal, id, or constant then EnterName() is     */
/* called to enter it into the nametable.                             */
/* Lastly, resets the buffer ptr to tokenbuff[0].                     */

{
extern   long  tokenbuff[], *ptr;
int      token_num;                             /* identifies a token type       */
NameRec  *nptr;                                 /* ptr to structure of NameRec   */

   ptr = start;                                 /* intit ptr to travel thru buff  */

   do
   {  token_num = GetToken(token);
      *ptr = token_num;
      ++ptr;

      switch (token_num)
      {  case LITERAL_    :
         case CONSTANT_   :
         case IDENTIFIER_ :
         {  token[0] = strlen(token);           /* insert length of sting        */
            if((nptr=EnterName(token)))
            {  *ptr = (long)nptr;               /* address of token              */
               ++ptr;
            }
            else ErrorHandler(NULL,ERR7,NULL);  /* HANDLE MEMORY OVERFLOW!       */
            break;
         }                                      /* end case                      */
         default:                               /* do nothing                    */
      }                                         /* end switch                    */
   } while((token_num != EOF)  &&
         (ptr < &tokenbuff[BUFSIZE]));

   ptr = &tokenbuff[0];                         /* reset the buffer ptr          */
 }                                              /* end FillBuff()                */
/******************************************************************************/
```

```
   long
ByPass(tgt)
   int   tgt;

/* Checks to see if the next token in the buffer matches the target.  */
/* If so, then returns the token no. and increments the buffer         */
/* pointer                                                             */
{
extern   long   tokenbuff[], *ptr;

   if(ptr >= &tokenbuff[BUFSIZE])                /* see if at end of buffer      */
      FillBuff(&tokenbuff[0]);                   /* refill buffer                */

   while(*ptr == EOLN_)
   {  ++ptr;                                     /* increment counter & skip     */
      ++line_no;
      if(ptr == &tokenbuff[BUFSIZE])             /* see if at end of buff        */
         FillBuff(&tokenbuff[0]);                /* refill buffer                */
  }                                              /* end while                    */

   if (*ptr != tgt)
      return(FALSE);

   ++ptr;                                        /* otherwise, it was found      */

  if(ptr == &tokenbuff[BUFSIZE])                 /* if at end of buffer          */
    FillBuff(&tokenbuff[0]);                     /* refill buffer                */

   switch (tgt)
   {  case LITERAL_     :
      case IDENTIFIER_  :
      case CONSTANT_    :                        /* return ptr to struct         */
         return(*(ptr++));                       /* holding the token            */

      default:                                   /* just return true             */
         return(tgt);
   }                                             /* end swithch                  */
}                                                /* end ByPass()                 */
/*********************************************************************/

   int
IsFormal(root)                                   /* root is ptr to subtree       */
    NodeRec   *root;                             /* currently working with       */

/* Required to make the language deterministic. Compound() returned   */
/* TRUE and "|->" was subsquently found. Formal is a proper subset of */
/* the compounds so need to insure no errors in the formals.          */
/* Performs a preorder search of the subtree. NOTE: assumes that root */
/* initially points to a non-null compound list.                      */
{

#ifdef   DEBUG
printf("isformal entered,root->name = %d\n",root->name);
if (root == NULL) printf("root is null\n");
#endif

   if(root == NULL)
      return(TRUE);

   if(root->name==COMMA_ || root->name==IDENTIFIER_
                         || root->name==ELLIST)
   if((IsFormal(root->lptr))
                  && (IsFormal(root->rptr)))
      return(TRUE);

   return(FALSE);
}                                                /* end Isformal                   */
/*********************************************************************************/

   int
IBall(tgt,index)
   int   tgt, index;
/* Checks to see if the (index)th token in the buffer matches the      */
/* target.  If it does returns TRUE else FALSE.  Does not increment    */
/* the buffer pointer.  Checks for full buffer implemented in this     */
/* manner to allow for future flexibility.  Could have used simple     */
/* heuristic of:                                                       */
/*       if(ptr + (3*index) > &tokenbuff[BUFSIZE])    RefilBuffer;     */
/*  at the expense of generality                                       */
{
extern  long   tokenbuff[], *ptr;
long    *tptr;

   if(ptr >= &tokenbuff[BUFSIZE])                /* see if at end of buff if       */
      FillBuff(&tokenbuff[0]);                   /* so, refill buffer              */

                                                 /* start over if had to refill    */
DO_AGAIN:                                        /* buffer during check for tgt    */
   tptr = ptr;                                   /* set working pointer            */

   while(*tptr == EOLN_)
   {  ++tptr;                                    /* increment tptr & skip EOLNs    */
      if(tptr == &tokenbuff[BUFSIZE])            /* see if at end of buff          */
        goto REFIL;                              /* nedd to refill buffer and      */
                                                 /* then start over                */
                                                 /* end while                      */
   for(;index >1; --index)                       /* only enter for loop if need to */
   {  switch (*tptr)                             /* look more than one char ahead  */
      {  case IDENTIFIER_:                       /* double skip because next       */
         case CONSTANT_:                         /* entry is addr of element       */
         case LITERAL_:   tptr += 2;  break;

         case EOLN_:
            while(*tptr == EOLN_)
            {  ++tptr;                           /* increment counter & skip       */
               if(ptr == &tokenbuff[BUFSIZE])
               goto REFIL;                       /* refill buffer & start over     */
            }                                    /* end while                      */
         default:          ++tptr;
      }                                          /* end switch                     */
      if(tptr >= &tokenbuff[BUFSIZE])            /* check if will overflow buff    */
         goto REFIL;
   }                                             /* end for                        */
   if (*tptr != tgt) return(FALSE);
   else   return(TRUE);

REFIL:                                           /* take what's left in buffer,    */
                                                 /* put at beginning, now refil    */
                                                 /* rest of buffer                 */
   for(tptr = &tokenbuff[0];
       ptr < &tokenbuff[BUFSIZE];  ptr++,tptr++)
      *tptr = *ptr;                              /* refill buffer from current     */
   FillBuff(tptr);                               /* posit to end                   */
   goto DO_AGAIN;                                  /* refilled buffer, so start      */
                                                   /* over                           */
}                                                  /* end IBall()                    */
/************************************************************************/

   char *
NodeName(ptr)
   NodeRec   *ptr;
/* Accepts a ptr to a structure of NodeRec.  Dereferences this node   */
/* to get a ptr to structure of NameRec which hold the string         */
/* containing the name of the value in NodeRec. Returns the name to   */
/* calling routine                                                    */
{
NameRec   *temp;                                   /* temp ptr to data struct        */
                                                   /* holding name of "*ptr"         */
   temp = (NameRec *)(ptr->index);
   return(temp->name + 1);
}                                                  /* end NodeName()                 */
/************************************************************************/
```

APPENDIX I

ROCK COMPILER — ERROR HANDLER

```
/**************************************************************************
*                              PUBLIC DOMAIN SOFTWARE                     *
* Name       :   Error Handler                                            *
* File       :   errors.c                                                 *
* Authors    :   Maj E.J. COLE / Capt J.E. CONNELL                        *
* Started    :   01/20/87                                                 *
* Archived   :   04/07/87                                                 *
* Modified   :                                                            *
***************************************************************************
* This file contains the execution modules for error recovery.            *
*                         ErrorHandler(), EatEm()                         *
*                                                                         *
* Algorithm :  ErrorHandler() is called by other modules in the           *
*              compiler.  It insures the error count is updated and       *
*              the* error is written to the error file.   If required,    *
*              ErrorHandler() calls EatEm() to gobble tokens to get to    *
*              a known point in  the parse.   Used during error           *
*              recovery.   After MAXERRORS number of errors simply        *
*              returns to calling routine.                                *
* NOTE      :  'errorfile' must have been initially created before        *
*              ErrorHandler() is first called - don't want to append      *
*              to last times errors!                                      *
***************************************************************************
* Modified   :                                                            *
*                                                                         *
**************************************************************************/

#include <stdio.h>
#include <scanner.h>
#include <errors.h>
extern   FILE   *errorfile;                         /* working file             */

int      num_errors = 0;                            /* running talley of # erors     */
                                                    /* found - global var            */
char      *errors[] = {                             /* array of error messages       */
/* 0 */           " incomplete '|->'",
/* 1 */           " RESERVED FOR FUTURE USE",
/* 2 */           "'\\' without following '/', logical OR is '\\/'",
/* 3 */           "'$' without following 'R','N','Z','B',or '1'",
/* 4 */           "invalid numeric constant ==> ",
/* 5 */           "literal without ending   - ",
/* 6 */           "unidentified char in input program ==> ",
/* 7 */           "MEMORY OVERFLOW DURING COMPILATION",
/* 8 */           "error in statement following ==> ",
/* 9 */           "error in type definition following ==> ",
/* a */           "unable to complete definition of blockbody after keyword LET",
/* b */           "missing or misplaced ';' after definition",
/* c */           "valid qualexp/exp not found in the def/auxdef",
/* d */              "valid typeexp not found in the def",
/* e */              "formals list missing or error in formals list",
/* f */              "misplaced or missing ",
/* g */              "at least one identifier must follow keyword TYPE",
/* h */              "unable to complete def/auxdef following keyword AND",
/* i */              "missing or invalid auxdef after keyword WHERE",
/* j */              "missing or misplaced closing paren in formals list",
/* k */              "error in processing multiple Actuals",
/* l */              "missing literal after keyword FILE",
/* m */              "missing or invalid exp following KEYWORD ",
/* n */              "IF statement w/o ENDIF",
/* o */              "error in formals preceding |->",
/* p */              "missing or invalid QualExp following COMMA operator",
/* q */              "error in ArgBinding - check QualExp or closing bracket",
/* r */              "OZONE LEVEL I - for 19.99 the feature can be implemented in 1999",
/* s */              " ",
/* t */              " ",
/* u */              " ",
/* v */              " ",
/* w */              " ",
/* x*/               " ",
/* y */              " ",
/* z */              " ",
/* aa */             " NUMERIC VALUE EXPECTED ",
/* bb */             " NATURAL EXPECTED ",
/* cc */             " INTEGER OR NATURAL EXPECTED ",
/* dd */             " ERROR IN TUPLE DEFINITION ",
/* ee */             " UNDEFINED VARIABLE IN AND SCOPE ",
/* ff */             " FUNCTION WITHOUT FUNCTION DEFINITION ",
/* gg */             " FORMALS MISMATCHED ",
/* hh */             " FUNCTION CALLED WITHOUT FUNCTION DEFINITION ",
/* ii */             " REAL NUMBER EXPECTED ",
/* jj */             " INVALID CONSTANT EXPRESSION ",
/* kk */             " BOOLEAN VALUE EXPECTED ",
/* ll */             " BOOLEAN OPERATOR EXPECTED ",
/* mm */             " OUT OF RUN-TIME MEMORY SPACE ",

         };

/*****************************************************************************/

   void
ErrorHandler(line_no,err_no,str_num)
   int   line_no, err_no;
   long   str_num;

/*   use long because str_num is either pointer to a string "long"     */
/*   or an actual number (int or long)                                  */

{

#ifdef   DEBUG
printf("eh entered, err# = %d, str_num = %ld\n",err_no,str_num);
#endif

   if (++num_errors > MAXERRORS)    return;

   errorfile = fopen("errors.phi","a");           /* append to what's there       */

   if (err_no == ERR7)                            /* no more memory -             */
   {  fprintf(errorfile,"%s\n",errors[err_no]);   /* get out and start over       */
      user_err()
   execl("rock.exe","rock.exe",NULL);
}                                                /* end if no more memory          */

fprintf(errorfile,"line %3d : %s ",
        line_no,errors[err_no]);

switch (err_no)    {
   case ERR4:
   case ERR5:   fprintf(errorfile,"%s\n",(char *)str_num);   break;

   case ERR6:   fprintf(errorfile,"%.1s\n",(char *)str_num); break;

   case ERR8:   switch(str_num)
   {  case LEQ_ :           fprintf(errorfile,"<=\n");       break;
      case NEQ_ :           fprintf(errorfile,"<>\n");       break;
      case GEQ_ :           fprintf(errorfile,">=\n");       break;
      case EQ_  :           fprintf(errorfile,"=\n");        break;
      case ADD_ :           fprintf(errorfile,"+\n");        break;
      case SUB_ :           fprintf(errorfile,"-\n");        break;
      case MULT_:           fprintf(errorfile,"*\n");        break;
      case IDIV_:           fprintf(errorfile,"%\n");        break;
      case RDIV_:           fprintf(errorfile,"/\n");        break;
      case SUBSCRIPT_:      fprintf(errorfile,"!\n");        break;
      case ORLOG_ :         fprintf(errorfile,"\\/\n");      break;
      case ANDLOG_:         fprintf(errorfile,"/\\\n");      break;
      case NEGLOG_:         fprintf(errorfile,"~\n");        break;
      case COLON_ :         fprintf(errorfile,":\n");        break;
      case CAT_   :         fprintf(errorfile,"^\n");        break;
      case LINERTARROW_:    fprintf(errorfile,"|->\n");      break;
      case (KW_+GREATER_):  fprintf(errorfile,"GREATER\n"); break;
      case (KW_+IN_)    :   fprintf(errorfile,"IN\n");       break;
      case (KW_+LESS_) :    fprintf(errorfile,"LESS\n");     break;
      case (KW_+NOTIN_):    fprintf(errorfile,"NOTIN\n");    break;
      default:              ;
         fprintf(errorfile,"UNDEFINED error\n");
   }                                             /* end switch case ERR8           */
      break;

   case ERR9:   switch(str_num)
   {  case ADD_    :   fprintf(errorfile,"+\n");   break;
      case MULT_   :   fprintf(errorfile,"*\n");   break;
      case RTARROW_:   fprintf(errorfile,"->\n");  break;
      case LTPAREN_:   fprintf(errorfile,"(\n");   break;
      default:         ;
         fprintf(errorfile,"UNDEFINED error\n");
   }                                             /* end switch case ERR9           */
      break;

   case ERR_f: switch(str_num)    {
      case KW_+AND_:
      case KW_+WHERE_:
         fprintf(errorfile,"==\n");
         break;
      case RTPAREN_:
         fprintf(errorfile,")\n");
         str_num=NULL; break;                    /* don't want to go to EatEm      */
      case RTSQUIG_:
         fprintf(errorfile,"}\n");
         str_num=NULL; break;                    /* don't want to go to EatEm      */
      case END_SEQUENCE_:
         fprintf(errorfile,">\n");
         str_num=NULL; break;                    /* don't want to go to EatEm      */
         case KW_+END_:
            fprintf(errorfile,"KEYWORD END\n");
            str_num += KW_; break;                 /* set up for call toEatEm       */
         case KW_+THEN_:
            fprintf(errorfile,"KEYWORD THEN\n");
            break;

         default:
            fprintf(errorfile,"UNDEFINED error\n");
         }                                         /* end switch case ERR_f         */
         break;

      case ERR_m: switch(str_num)   {
         case IF_    :
            fprintf(errorfile,"IF\n");      break;
         case ELSIF_ :
            fprintf(errorfile,"ELSIF\n");   break;
         case ELSE_  :
            fprintf(errorfile,"ELSE\n");    break;
         case THEN_  :
            fprintf(errorfile,"THEN\n");    break;
         case BEGIN_ :
            fprintf(errorfile,"BEGIN\n");   break;
         default:
            fprintf(errorfile,"UNDEFINED error\n");
         }                                         /* end switch case ERR_m         */

         str_num += KW_;                           /* set str_num up to be passed   */
         break;                                    /* to EatEm()                    */
      default:   fprintf(errorfile,"   \n");
   }                                               /* end switch                    */

   fclose(errorfile);

   if ((err_no >= ERR_a) &&
       (err_no < ERR_aa) &&
       (str_num != NULL))
      EatEm((int)str_num);
}                                                  /* end ErrorHandler              */
/*************************************************************************************/

   void
EatEm(tgt)
   int tgt;

/* Increments token buffer pointer until tgt token is found.             */
/* Use in error recovery to reach a known point in the program.          */
{
extern   long   tokenbuff[], *ptr;
extern   int    line_no;

#ifdef   DEBUG
printf("eatem entered, tgt = %d\n",tgt);
#endif

   while(*ptr != EOF_)   {
      switch (tgt)
      {  case EOLN_ :
            ++ptr;   ++line_no;   break;

         case SEMI_ :
            if((*ptr==SEMI_) || (*ptr==KW_+LET_))
      return;
   ++ptr;   break;

case EQUIV_:      switch ((int)*ptr)
{  case EQUIV_    :
   case SEMI_     :
   case KW_+AND_  :
   case KW_+LET_  :   return;
   default:           ++ptr;
   }    break;                          /* end switch case EQUIV         */

case KW_+WHERE_  :   switch ((int)*ptr)
{  case KW_+WHERE_:
   case KW_+AND_  :
   case KW_+LET_  :
   case SEMI_     :   return;
   default        :   ++ptr;
   }    break;                          /* end switch case WHERE         */

case KW_+AND_    :   switch ((int)*ptr)
{  case KW_+AND_  :
   case KW_+LET_  :
   case SEMI_     :   return;
   default        :   ++ptr;
   }    break;                          /* end switch case AND           */

case RTPAREN_    :   switch ((int)*ptr)
{  case RTPAREN_  :
   case LTPAREN_  :
   case COMMA_    :
   case EQUIV_    :
   case LINERTARROW_:
   case KW_+LET_  :
   case KW_+AND_  :
   case SEMI_     :    return;
   default        :    ++ptr;
   }    break;                          /* end switch case RTPAREN       */

case KW_+ IF_     :
case KW_+ ELSIF_  :
case KW_+ ELSE_   :
case KW_+ THEN_   :    switch((int)*ptr)
{  case KW_+ ELSIF_  :
   case KW_+ ELSE_   :
   case KW_+ ENDIF_  :
   case KW_+ THEN_   :  return;
}                                       /* end switch case THEN, etc     */
   ++ptr;   break;

case COMMA_          :  switch ((int)*ptr)
{  case COMMA_       :
   case LTPAREN_     :
   case RTPAREN_     :
   case LTSQUIG_     :
   case RTSQUIG_     :
   case ST_SEQUENCE_ :
   case END_SEQUENCE_:
   case SEMI_        :
   case KW_+LET_     :
   case KW_+WHERE_   :
   case KW_+ AND_    :  return;
   default           :  ++ptr;
```

```
        }   break;                                      /* end switch case COMMA          */

        case KW_+END_         :
        case KW_+BEGIN_       :  switch ((int)*ptr)
        {  case KW_+END_      :
           case KW_+LET_      :
           case KW_+WHERE_    :
           case KW_+AND_      :
           case COMMA_        :
           case RTPAREN_      :
           case RTSQUIG_      :
           case END_SEQUENCE_:
           case SEMI_         :   return;
           default            :   ++ptr;
        }   break;                                      /* end switch case BEGIN/END      */

        default :
           return;
     }                                                  /* end swithch                    */
  }                                                     /* end while                      */
}                                                       /* end EatEm()                    */
/***********************************************************************************/
```

APPENDIX J

ROCK COMPILER — SEMANTIC CHECKER

```
/**********************************************************************
* PUBLIC DOMAIN SOFTWARE                                              *
*                                                                     *
* Name      :  Semantic Checker Module 0                              *
* File      :  Sem0.c                                                 *
* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :  02/01/87                                               *
* Archived  :  04/03/87                                               *
* Modified  :                                                         *
***********************************************************************
* This file contains the following modules for the PHI parser:       *
*                                                                     *
*       Hnumconvert             Numconvert                            *
*                                                                     *
* Algorithm :                                                         *
*     This module contains procedures for type conversion.  If the    *
* rt child of a node may be converted to the lt type but the con-     *
* verse is not true, "Hnumconvert" is called.  If either side may be  *
* converted, "numberconvert" is called                                *
***********************************************************************
* Modified  :                                                         *
**********************************************************************/

/****************************** Externals ****************************/
#include <semcheck.h>

extern void terror ();
/********************** hnumconvert ******************************/
   PHITYPE
hnumconvert (ltype, rtype, ptr)                    /* Type conversions for the       */
                                                   /* right side of the tree only    */
   PHITYPE ltype, rtype;                           /* Left and Right types           */
   nodal ptr;                                      /* Ptr to the root working with   */
{extern void c_ztor ();                            /* Generates code to convert      */
                                                   /* integer/natural to real        */

    if ((ltype == BOOLEAN) && (rtype == BOOLEAN))
       return (BOOLEAN);                           /* No type conversion needed      */

    switch (ltype)  {                              /* Predicate actions on type of lt*/
       case (REAL) : switch (rtype) {              /* side of node                   */

           case (REAL) : return (REAL);            /* Matching types; no conv req    */
           case (INTEGER) :
           case (NATURAL) :                        /* Generate code for conversion   */
               c_ztor ();
               return (REAL);
           default :
                terror (ERR_aa, ptr->ln);           /* No appropriate match; error    *
                return (REAL);     }                /* Rtn real so semantic check cont*

        case (INTEGER) : switch (rtype) {
            case (INTEGER) :
            case (NATURAL) : return (rtype);        /* Matching types, no conv req    */
            default :
                terror (ERR_cc, ptr->ln);           /* Can't convert from real to int */
                return (INTEGER); }                 /* so sandbag the programmer      */

        case (NATURAL) :
            if (rtype == NATURAL)
                return (rtype);                     /* Only one match poss w/o error  */
            else {
                terror (ERR_bb, ptr->ln);
                return (NATURAL);
                }
        default : terror (ERR_aa, ptr->ln);
            return (NATURAL);
    }
}

/************************ Numconvert ****************************/
    PHITYPE
numconvert (ptr)                                    /* Do number conversions for      */
                                                    /* both left and right side       */
    nodal ptr;
{PHITYPE ltype, rtype;                              /* Left and right child types     */
 extern PHITYPE semcheck ();
 extern void c_ztor ();

    ltype = semcheck (ptr->lptr);                   /* Get left type                  */

    if (ptr->rptr->name == (KW_ + ENDIF_))          /* Special case of "if" sequence  */
        return (ltype);
    rtype = semcheck (ptr->rptr);                   /* Get right type                 */

    if ((ltype == BOOLEAN) && (rtype == BOOLEAN))   /* No conversion necessary        */
        return (BOOLEAN);

    switch (ltype) {                                /* Predicate actions on lt type   */
        case (REAL) : switch (rtype) {
            case (REAL) : return (REAL);            /* Types are same; no action req  */
            case (INTEGER) :
            case (NATURAL) :                        /* Generate code for int/nat      */
                c_ztor ();                          /* to real conversion             */
                return (REAL);
            default :                               /* No converison possible         */
                terror (ERR_aa, ptr->rptr->ln);
                return (REAL);
        }
        case (NATURAL) : switch (rtype) {
            case (REAL) :                           /* Convert left side              */
                c_ztor ();
                return (REAL);
            case (INTEGER) :
                return (INTEGER);                   /* No conversion necessary        */
            case (NATURAL) :
                return (NATURAL);                   /* No conversion necessary        */
            default :
                terror (ERR_aa, ptr->rptr->ln);
                return (NATURAL);
        }

        case (INTEGER) : switch (rtype) {
           case (REAL) :                               /* Convert left side            */
              c_ztor ();
              return (REAL);
           case (INTEGER) :
           case (NATURAL) :
              return (INTEGER);                        /* No conversion necessary      */
           default :
              terror (ERR_aa, ptr->rptr->ln);
              return (NATURAL);
        }
        default :
           terror (ERR_aa, ptr->lptr->ln);             /* Types are not numeric        */
           return (NATURAL);
     }
  }
```

```
/**********************************************************************
* PUBLIC DOMAIN SOFTWARE                                              *
*                                                                     *
* Name      :  Semcheck Module 1                                      *
* File      :  Sem1.c                                                 *
* Authors   :  Maj  E.J. COLE / Capt J.E. CONNELL                     *
* Started   :  01/02/87                                               *
* Archived  :  01/10/87                                               *
* Modified  :                                                         *
***********************************************************************
* This file contains the following modules for the PHI parser:        *
*                                                                     *
*       Tletdef              Trtarrow          Tkindef                *
*       Twhere               Tdataauxdef       Tauxand                *
*       Tandcheck            Tauxand           Ttypetimes             *
*                                                                     *
* Algorithm :                                                         *
*       This module contains scoping procedures (Twhere and Tauxand)  *
* definition procedures (trtarrow, tkindef, ttypetimes) and the data  *
* definition procedure.                                               *
*                                                                     *
***********************************************************************
* Modified  :                                                         *
**********************************************************************/

/****************************** Externals ****************************/
#include <semcheck.h>
#include <string.h>                                /* For "strcpy"               */

extern int typeptr;                                /* Typetable and pointer      */
extern tnode types [];
extern void terror ();

fnode *fhead = NULL;
/************************* Tletdef ********************************/
    void
tletdef (ptr)                                      /* checks types of both branches  */
    nodal ptr;
{
    semcheck (ptr->lptr);
    semcheck (ptr->rptr);
}

/************************* Trtarrow *******************************/
    PHITYPE
trtarrow (ptr)                                     /* Returns type               */
    nodal ptr;
{PHITYPE ltype, rtype;
 extern void putform ();

    ltype = semcheck (ptr->lptr);                  /* Check left side type       */
   rtype = semcheck (ptr->rptr);                   /* Check right side type      */

    if (!(ptr->lptr->name == TYPETIMES) ||
         (ptr->lptr->name == TYPEPLUS))
    putform (ltype);                               /* Only if leftnode not '*' or '+'*/

    return (rtype);
}
```

```
/************************* Tkindef ******************************/
    void
tkindef (ptr)                                          /* Adds variable name to defstack */
    nodal ptr;
{extern defptr defhead;
 extern void putdef ();
 PHITYPE rtype;

    rtype = semcheck (ptr->rptr);
    putdef (rtype, ptr->lptr);                         /* Put definition in defstack     */
    defhead->fptr = fhead;                             /* Append formal types to entry   */
   fhead = NULL;                                       /* Kill fhead                     */
}

/************************* Twhere *******************************/
    PHITYPE
twhere (ptr)                                           /* Semcheck where node            */
    nodal ptr;
{PHITYPE type;

    semchecker (ptr->lptr);                            /* Check leftside                 */
    type = semchecker (ptr->rptr);                     /* Check right side               */
    return (type);
}

/************************ TDatauxdef ****************************/
    void
tdatauxdef (ptr)                                       /* WORKS FOR ONE FORMALS ONLY     */
    nodal ptr;
{extern void c_store_code (), c_jmp ();
 extern PHITYPE getdtype ();
 extern defptr finddef ();
 extern char *name ();
 defptr d_ptr;
 char   *holder = malloc (8),                          /* Temp holder for function name  */
        *nme = malloc (8);
 PHITYPE rtype,                                        /* Type of left and right nodes   */
         type,                                         /* Type of datadef                */
         count = 0;

    nme = strcpy (nme, name ());
    c_jmp (nme);

    holder = strcpy (holder, name());                  /* Calculate function name        */
    c_start_proc (holder);                             /* Gen code for starting proc     */
    rtype = semcheck (ptr->rptr);                      /* Get type of right ptr          */

    if (ptr->lptr->name == IDENTIFIER_) {              /* Open can of worms to typecheck */
                                                       /* if left is ident.              */
        if(!(d_ptr=finddef(ptr->lptr->index))) {       /* No prev decl of this variable  */
            ptr->lptr->type = rtype;
            putvar (rtype, ptr->lptr);
                }
    else if (d_ptr->fptr == NULL)  {                   /* Prev decl of var is data def   */
        ptr->lptr->type = getdtype (d_ptr);
        type = hnumconvert (ptr->lptr->type,
        rtype, ptr);                                   /* Convert rt type if feasible    */
        putvar (type, ptr->lptr);
                }
    else                                               /* Prev decl of var is another var*/
        terror (ERR_dd, ptr->lptr->ln);
                }
    while (*(holder + count) != NULL) {           /* Push piano through the door    */
                                                  /* to copy strings                */
        (ptr->lptr->label [count]) = (*(holder + count));
        ++count;                       }

    c_store_code ("ret\n");                       /* Generate code to end procedure */
    c_store_code (nme);                           /* CANNOT USE C_END_PROC () HERE; */
                                                  /* NO SCOPE CHANGE!               */
    c_store_code (":\n");
}

/**************************** And_Check ******************************/
    void
and_check (mark, ptr, mark_and)                   /* Check and_list for var defs    */
    varptr mark;                                  /* Scope delimiter                */
    and_ptr *mark_and, ptr;
{extern varptr varhead;
 extern int buff_ptr;
 extern char *code_buffer;
 int buff_holder;
 varptr v_ptr = varhead;

    if (ptr != NULL) {                            /* Ptr = NULL is base for recurs  */
        and_check (mark, ptr->link, mark_and);    /* of and_check                   */
        do {                                      /* Loop to evaluate all proper    */
                                                  /* varptr entries                 */
                                                  /* Check if equal names in        */
                                                  /* and_list & var_list            */
                                                  /* Not a function definition      */
            if(v_ptr->nptr->index==ptr->ptr->index){
                buff_holder = buff_ptr;           /* Save code buffer pointer       */
                buff_ptr = ptr->buffptr;          /* Get location of variable code  */
                c_call_proc (v_ptr->nptr->label); /* Generate code                  */
                buff_ptr = buff_holder;           /* Restore buffer pointer         */

                if (*mark_and == ptr)             /* Traverse list                  */
                *mark_and = ptr->link;

                del_and (ptr);
                break; }

            if (v_ptr == mark) break;             /* End of var list reached        */

            v_ptr = v_ptr->link;
        } while (TRUE);                           /* Exit is accomplished using a   */
                                                  /* break in the loop              */
    }
}

/****************************** Tauxand ******************************/
    void
tauxand (ptr)                                     /* Semantic check for and node    */
    nodal ptr;
{extern FLAG and_flag;
 extern and_ptr and_head;
 int save_and;                                    /* Holder for and flag            */
 varptr mark;                                     /* Mark top entry in the varlist  */
 and_ptr tptr, mark_and = and_head;               /* Mark current head of and_stack */

    save_and = and_flag;                          /* Save current and_flag          */
    and_flag = TRUE;                              /* Set and_flag                   */
    semcheck (ptr->lptr);                              /* Semantic Check                  */
    mark = varhead;
    semcheck (ptr->rptr);

    and_check (mark, and_head, &mark_and);             /* Check all new fctn & data defs */

    and_flag = save_and;                               /* Restore and flag                */

    tptr = and_head;

    while (tptr != NULL)                               /* Traverse list until end         */
       tptr = tptr->link;

    if (mark_and != and_head)                          /* Undefine variables found        */
    terror (ERR_ee, ptr->ln);

/****************************** TTypeTimes ****************************/
    PHITYPE
ttypetimes (ptr)                                       /* Semantic check '*' when used    */
                                                       /* for types                       */
    nodal ptr;
{extern void putform ();
 PHITYPE type;

    putform (semcheck (ptr->lptr));                    /* Attach formal type to           */
                                                       /* formal list                     */
    if (type = semcheck (ptr->rptr))                   /* Look for right type; if 0,      */
                                                       /* end of insertions               */
    putform (type);

    return (NULL);                                     /* Always return NULL;             */
                                                       /* This value is used by parent    */
}
```

```
/************************************************************************
* PUBLIC DOMAIN SOFTWARE                                                *
*                                                                       *
* Name      :  Semcheck Module 2                                        *
* File      :  Sem2.c                                                   *
* Authors   :  Maj  E.J. COLE / Capt J.E. CONNELL                       *
* Started   :  01/02/87                                                 *
* Archived  :  04/10/87                                                 *
* Modified  :                                                           *
*************************************************************************
* This file contains the following modules for the PHI parser:          *
*                                                                       *
*       Matchfor                Tfunauxdef              Tfunid          *
*       Tactualist              Tid                     Act_Walk        *
*       Telist                                                          *
*                                                                       *
* Algorithm :                                                           *
*    This module contains the procedures needed to define and call      *
* functions. Tfunauxdef will set up the run-time structure of the fun-*
* ction, tfunid will check the semantics of the function, & matchfor, *
* called by tfunid, checks for the proper type & number of formal pa- *
* rameters.                                                             *
*    Tactualist coordinates the checking of a function call.  It uses *
* both telist and act_walk.  Actwalk determines whether the number &  *
* type of actuals is correct, and telist checks each element list and *
* returns its type.                                                     *
*        Tid performs semantic checking for program variables.          *
*                                                                       *
*************************************************************************
* Modified  :                                                           *
************************************************************************/

/*************************** Externals **********************************/
#include <semcheck.h>
#include <string.h>                              /* For "strcpy"           */

extern tnode types [];
extern varptr varhead;
extern void terror (), c_store_code ();

/***************************** Globals **********************************/
int actual_count = 0;                            /* count of all actuals   */

/***************************** Matchfor *********************************/
   FLAG
matchfor (nptr, def)                             /* Match formals          */
                                                 /* Called by tfunid () only */
   nodal nptr;                                   /* Ptr to rt side of funid node */
   defptr def;                                   /* Ptr to var table for func name */
{extern long curr_addr;
 extern fnode *getfptr ();
 extern FLAG form;                               /* Flag set when formals  */
                                                 /* are generated          */
 fnode *tptr = getfptr (def);

   form = TRUE;
   tptr = def->fptr;
   curr_addr = 0;
```

```
    if (nptr->name == IDENTIFIER_) {                    /* Only one formal              */
        (nptr->type) = tptr->type;
            nptr->addr = curr_addr;
        putvar (tptr->type, nptr);
        nptr = nptr->rptr;
        tptr = tptr->link;
    }

    else {                                              /* Multiple formals             */

        do {
            nptr->lptr->type = tptr->type;
            nptr->lptr->addr = curr_addr;
            curr_addr = curr_addr +
                types [tptr->type].bytes;
            putvar (tptr->type, nptr->lptr);
            nptr = nptr->rptr;
            tptr = tptr->link;
        }   while((nptr!=NULL)&&(tptr!=NULL));          /* Halt when end reached        */
                                                        /* by either ptr                */
    }

    form = FALSE;

    if (nptr != NULL || tptr != NULL)                   /* One ptr isn't at end of run  */
        return (FALSE);                                 /* Error handled in calling fctn */

    else return (TRUE);
}

/********************* Tfunauxdef ******************************/
    void
tfunauxdef (ptr)                                        /* Type check funauxdef         */
    nodal ptr;
{extern long curr_addr;
 extern void c_end_proc (), c_jmp ();
 extern char *name ();
 extern nodal hnumconvert ();
 char *nme = malloc (8);
 PHITYPE ltype, rtype;
 varptr var1, mark = varhead;
 long pres_addr = curr_addr;


    nme = strcpy (nme, name ());                        /* Name for jump around function */
    c_jmp (nme);                                        /* Gen code to jump around fctn  */

    ltype = semcheck (ptr->lptr);
    rtype = semcheck (ptr->rptr);

    while (varhead->link != mark) {                     /* Eliminate formals from lnk lst */
        var1 = varhead;
        varhead = varhead->link;
        var1->link = NULL;
        free (var1);        }

    ptr->rptr =                                         /* Convert if needed            */
    hnumconvert (ltype, rtype, ptr->rptr);
    c_end_proc (nme);

    curr_addr = pres_addr;                              /* Reset addresses              */
}
```

```
/****************************** Tfunid *********************************/
   PHITYPE
tfunid (ptr)                                  /* Semantic Check for tfunid      */
   nodal ptr;
{extern defptr finddef ();
 extern long curr_addr;
 extern char *name ();
 int count = 0;                               /* Generic loop varient           */
 defptr def;
 char *holder = malloc (8);

   if (!(def = finddef (ptr->lptr->index))) {    /* Func name not found         */
      terror (ERR_ff, ptr->ln);
      return (NOTFOUND);        }

   else {
     ptr->lptr->type = def->type;             /* Set node type                  */
     ptr->type = def->type;
     putvar (ptr->lptr->type, ptr->lptr, FALSE);

     if (!matchfor (ptr->rptr, def))          /* Match formals                  */
      terror (ERR_gg, ptr->ln);

     else {
       holder = strcpy (holder, name ());

       while (*(holder + count) != 0) {       /* Push piano -> door to copy     */
                                              /* string to array                */
          (ptr->lptr->label [count])  =
          (*(holder + count));
             ++count;                   }

       ptr->lptr->addr = 0;

       c_start_proc (ptr->lptr->label);       /* Gen code for begin function    */
         }
      }
   return (ptr->type);
}

/*************************** Tellist ***********************************/
   void
telist (ptr)                                  /* Semantic Check for element lst */
   nodal ptr;
{
  if (ptr->rptr != NULL)                      /* Only semcheck if there is      */
                                              /* something there                */
     semcheck (ptr->rptr);

  semcheck (ptr->lptr);

  c_store_code ("call ppop\n");               /* Generate code                  */
  c_store_code ("push cx\n");
  c_store_code ("push di\n");
  ++actual_count;
}

/**************************** Act_Walk ********************************/
   void
act_walk (ptr,fptr)                           /* Recursive procedure to         */
                                              /* sem check actual list          */
   nodal ptr;
   fnode *fptr;
{

   if (ptr->rptr != NULL)                          /* Recurse until NULL ptr is hit  */
      act_walk (ptr->rptr, fptr->link);

   semcheck (ptr->lptr);
   if (ptr->lptr->name != ELLIST)    {
           ++actual_count;                         /* Incr count only if left        */
                                                   /* sibling is an ID               */
           c_store_code ("call ppop\n");           /* Generate code to put addresses */
                                                   /* on the stack                   */
           c_store_code ("push cx\n");
           c_store_code ("push di\n");
                   }
}
/********************************** Tactuals *************************************/
   PHITYPE
tactuals (ptr)                                     /* Evaluate actualists            */
   nodal ptr;
{extern void c_call_proc ();
 extern FLAG and_flag;
 extern varptr findvar ();
 extern defptr finddef ();
 extern char *name ();
 defptr def = finddef (ptr->lptr->index);          /* Defstack pointer               */
 varptr var = findvar (ptr->lptr->index);          /* Varstack pointer               */
 int count_hold = actual_count;
 char *long_buff = malloc (10);                    /* Buffer for long to string conv */
 long convert;                                     /* Conversion variable            */
 fnode *fptr;

   actual_count = 0;

   if (def) {                                      /* Definition found               */

       if ((!var && and_flag) || var)              /* Legitimate cases               */
       {
              fptr = def->fptr;                    /* Get a ptr to the formal nodes  */
              act_walk (ptr->rptr, fptr);
              convert = actual_count;
              c_store_code ("mov bx, ");           /* Generate code to put # of      */
                                                   /* actuals on the stack           */
              stcl_d (long_buff, convert);         /* Long to string conversion      */
              c_store_code (long_buff);
              c_store_code ("\n");
              c_call_proc ("i_mov");

           if ((and_flag) && (!var)) {             /* Cover "and" scoping rules      */
              add_and (ptr->lptr);
              c_call_proc (name ());               /* Holder for real name           */
                    }
              else
              c_call_proc (var->nptr->label);      /* Gen code to call function      */
              actual_count = count_hold;           /* Restore actual count           */
              return (def->type);
        }
      }
   terror (ERR_hh, ptr->ln);                       /* Function name not found        */
   return (NOTFOUND);
}
```

```
/**************************** Tid **********************************/
   PHITYPE
tid (ptr)                                           /* Typecheck Id node         */
   nodal ptr;
{extern void c_i_form ();
 extern long curr_addr;
 extern char *name ();
 extern int formal ();
 extern FLAG and_flag;
 extern varptr findvar ();
 extern defptr finddef ();
 char *long_buff = malloc (10);                     /* Buffer for long to string conv */
 varptr var = findvar (ptr->index);                 /* Look for definition of var    */
 defptr def;


   if (!var)           {                            /* Rtn type if var found         */
                                                    /* in def table                  */
     if (def = finddef (ptr->index)) {
        if (and_flag) {
          add_and (ptr);
          c_call_proc (name ());
          return (getdtype (def));                  /* Get and return type definition */
             }              }
      else return (NOTFOUND);
          }

   else if (formal (var))  {
       stcl_d (long_buff, var->nptr->addr);         /* Long to string conversion     */
      c_i_form (long_buff);
          }
   else                                             /* If no formal list, assume var  */
                                                    /* is an assignment              */
      c_call_proc (var->nptr->label);               /* Generate code to call procedure
   */
                                                    /* to assign value               */
   return (getvtype (var));                         /* Return variable type          */
}
```

```
/**********************************************************************
* PUBLIC DOMAIN SOFTWARE                                              *
*                                                                     *
* Name      :  Semcheck Module #3                                     *
* File      :  Sem3.c                                                 *
* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                      *
* Started   :  01/02/87                                               *
* Archived  :  04/02/87                                               *
* Modified  :                                                         *
***********************************************************************
* This file contains the following modules for the PHI parser:        *
*       Trdivide                Tidivide                Tarithop      *
*       Tprimary                Tconvert                Tconstant     *
*       Tand                    Tor                     Tnegation     *
*                                                                     *
* Algorithm :                                                         *
*      This module contains the procedures necessary for implementing *
* arithmetic & boolean operators.  Tarithop coordinates the semantic  *
* checking of arithmetic ops by calling the proper function based     *
* on the operator type.  Trdivide & Tidivide handle semantic checking *
* for real & int division, respectively.  For all other arithmetic    *
* ops, the numconvert procedure (sem0)is called to perform seman-     *
* tic checking, then code is generated.                               *
*     For each boolean operator, the appropriate child(ren) is checked*
* and code is generated for the operation.                            *
*     In addition, tconstant checks the type of a simple constant by  *
* calling convert, & then returns either the constant type or an error*
*                                                                     *
***********************************************************************
* Modified  :                                                         *
***********************************************************************
/*************************** Externals ********************************/
#include <semcheck.h>
#include <string.h>                           /* For "strcmpi"          */

extern void terror ();
extern void c_store_code ();                  /* Store asm language output */
                                              /* to a buffer              */

/************************* Trdivide ******************************/
   void
trdivide (ptr)                                /* Division of real operands */
   nodal ptr;
{PHITYPE ltype, rtype;
 extern FLAG err_found;
 extern void c_ztor ();

   ltype = semcheck (ptr->lptr);              /* Check left side for type  */

   switch (ltype) {                           /* Make convs or locate errors */
      case (REAL) : break;
      case (INTEGER) :
      case (NATURAL) :
           c_ztor ();
           break;
      default : terror (ERR_aa, ptr->lptr->ln);  /* Lt child must rtn numeric type */
           return;                            /* Error, no need to go thru acode*/
            }
   rtype = semcheck (ptr->rptr);                  /* Check right side for type      */

   switch (rtype) {
      case (REAL) : break;
      case (INTEGER) :
      case (NATURAL) :
          c_ztor ();
          break;
      default : terror (ERR_aa, ptr->rptr->ln);
           return;                                /* Error, no need to go thru acode*/
        }

   acode (ptr, REAL);                             /* Generate code                  */
}

/************************* TIdivide ******************************/
    PHITYPE
tidivide (ptr)                                    /* Semcheck for integer division  */
    nodal ptr;
{PHITYPE ltype, rtype, type = NATURAL;

   ltype = semcheck (ptr->lptr);                  /* TypeCheck both sides           */
   rtype = semcheck (ptr->rptr);

   switch (ltype) {                               /* Check lt for Int/Natural Type  */
      case (INTEGER) : type = INTEGER;
      case (NATURAL) :break;
      default : terror (ERR_cc, ptr->lptr->ln);   /* If not Int or Nat, error       */
                return (INTEGER);
             }

   switch (rtype) {                               /* Check rt for Int/ Natural type */
      case (INTEGER) : type = INTEGER;
      case (NATURAL) : break;
      default : terror (ERR_cc, ptr->rptr->ln);   /* If not Int or Nat, error       */
           return (INTEGER);
         }

   acode (ptr, type);                             /* Generate code                  */

   return (type);
}

/************************* TArithop *****************************/
    PHITYPE
arithop (ptr)                                     /* Type Check Addition,           */
                                                  /* Multiplication, Sequence Ops   */
    nodal ptr;
{extern PHITYPE numconvert ();
 int type;

   switch (ptr->name) {
     case (ADD_) :                                /* Addition falls through         */
     case (SUB_) :                                /* Subtraction falls through      */
     case (MULT_) :   if(type = numconvert(ptr)) {
                         acode (ptr, type);
                         return (type);}
                      else {
                         terror (ERR_aa, ptr->ln);
                         return (NATURAL);
                         }
     case (RDIV_) : trdivide (ptr);
           ptr->type = type;
           return (REAL);
     case (IDIV_) : tidivide (ptr);
           ptr->type = type;
           return (INTEGER);

     case (COLON_) : break;                         /* Dummies for now,               */
                                                    /* but watch our smoke!!!         */
     case (CAT_) :  break;                          /* "         "   "                */
           }
}

/************************* Tprimary *****************************/
    PHITYPE
tprimary (ptr)                                      /* Handle unary "+" or "-"        */
    nodal ptr;
{PHITYPE type;

   type = semcheck (ptr->rptr);

   if ((type != INTEGER) &&
      (type != REAL) &&
      (type != NATURAL))                            /* Check type of right node       */
      terror (ERR_aa, ptr->rptr->ln);               /* Type must be a number          */

   else if ((ptr->name) == NEG_) {                  /* Negate operation               */
       c_store_code ("call igetvalue\n");           /* Spew code                      */
       c_store_code ("neg ax\n");
       c_store_code ("call iputvalue\n");
               }
   return (type);                                   /* Note that no action is req     */
                                                    /* for unary "+"                  */
}

/************************* Convert ******************************/
    PHITYPE
convert (string)                                    /* Convert const to real, boolean,*/
                                                    /* or integer value               */
    stg string;                                     /* String to convert              */
{FLAG e = FALSE,                                    /* True if "e" or "E" read        */
      period = FALSE;                               /* True if a period has been read */
 int count = 0;                                     /* Garden variety loop counter    */

   if ((strcmpi (string, "FALSE")
      && strcmpi (string, "TRUE"))) {               /* If not boolean                 */


   while (string [count] != 0) {                    /* Loop until end of string       */
      if (!isdigit (string [count])) {              /* If character is not a digit    */

         if ((string [count] == 'e') ||
           (string [count] == 'E')) {               /* "e" or "E" found               */

            if (e) return (ERROR);                  /* Cannot have two "e"s           */
            else {
                 e = TRUE;

             if ((string [count + 1] == '+') ||     /* "+" or "-" character           */
                 (string [count + 1] == '-'))
                     ++count;
                  }                   }
         else if (string [count] == '.') {         /* Decimal point found             */
                 if (period) return (ERROR);       /* Cannot have two periods         */
                 else period = TRUE;
                    }

         else return (ERROR);         }

         ++count;     }

   if (e || period) return (REAL);                 /* If gauntlet has been run,       */
                                                   /* period or "e" makes real        */
   if (string [0] == '-')  return (INTEGER);       /* Negative sign makes an integer */

   return (NATURAL);          }                    /* If no other num types, natural */

   return (BOOLEAN);                               /* If not a number, a boolean      */
}

/************************* TConstant ******************************/
    PHITYPE
tconstant (ptr)                                    /* Handle constant nodes           */
    nodal ptr;
{extern put_addr ();
 PHITYPE type;                                     /* Constant type                   */
 NameRec *tptr;                                    /* Constant name                   */

   tptr = ptr->index;

   if (type = convert (tptr->name + 1)) {          /* Calculate type                  */
     ptr->type = type;
     put_addr (ptr, type);                         /* Fill node & increment address   */
     c_i_const (tptr->name + 1);
     return (type); }

   terror (ERR_jj, ptr->ln);                       /* No legitimate constant found    */
}
/******************************* Tand ********************************/
    PHITYPE
tand (ptr)                                         /* Sem Check for bool and node     */
    nodal ptr;
{PHITYPE ltype, rtype;

   ltype = semcheck (ptr->lptr);
   rtype = semcheck (ptr->rptr);

   if (!(ltype == BOOLEAN && rtype == BOOLEAN))    /* Both children must be boolean   */
      terror (ERR_kk, ptr->ln);

   c_store_code ("call land\n");                   /* Generate code                   */
   return (BOOLEAN);
}
/******************************* Tor *********************************/
    PHITYPE
tor (ptr)                                          /* Sema Check for bool or node     */
    nodal ptr;
{PHITYPE ltype, rtype;

   ltype = semcheck (ptr->lptr);
   rtype = semcheck (ptr->rptr);

   if (!(ltype == BOOLEAN && rtype == BOOLEAN))    /* Both children must be boolean   */
      terror (ERR_kk, ptr->ln);
   c_store_code ("call lor\n");                    /* Generate code                  */

   return (BOOLEAN);
}

/****************************** Tnegation ******************************/
    PHITYPE
tnegation (ptr)                                    /* Sema check for neg operation   */
    nodal ptr;
{
  if (!(semcheck (ptr->rptr) == BOOLEAN))          /* Rt child must be a boolean;    */
                                                   /* lt child is null               */
      terror (ERR_kk, ptr->ln);

  else c_store_code ("call negation\n");           /* Gen code for boolean negation  */

  return (BOOLEAN);
}
```

```
/***********************************************************************
* PUBLIC DOMAIN SOFTWARE                                               *
*                                                                      *
* Name      :  Semcheck Module #4                                      *
* File      :  Sem4.c                                                  *
* Authors   :  Maj  E.J. COLE / Capt J.E. CONNELL                      *
* Started   :  01/29/87                                                *
* Archived  :  04/03/87                                                *
* Modified  :                                                          *
************************************************************************
* This file contains the following modules for the PHI compiler:       *
*                                                                      *
*       Tif             Tthen           Telseif                        *
*       Telse           Tcomp                                          *
*                                                                      *
* Algorithm :                                                          *
*     This module contains the procedures necessary to implement the   *
* "if-then-elseif-else" series of commands.  Tif coordinates the seman-*
* tic checking by calling Tthen to check its left nodes, then calling  *
* telse to check its right nodes.  Telse will be called until the right*
* subtree runs out of "elses" and "elseifs".                           *
*                                                                      *
************************************************************************
* Modified  :                                                          *
***********************************************************************/

/**************************** Externals *******************************/

#include <semcheck.h>
#include <string.h>                               /* For "strcpy"                    */

extern FLAG err_found;
extern PHITYPE semcheck ();

extern char *name ();
extern void terror (), c_store_char ();

/**************************** Globals ****************************/
char *if_label = NULL;

/**************************** Tif ********************************/
   PHITYPE
tif (ptr)                                         /* Semantic checker for "if" node */
   nodal ptr;                                     /* Ptr to the node                 */
{extern PHITYPE numconvert ();                    /* Int, Natural to real converter */
 PHITYPE type;                                    /* Return value type               */

   if (if_label == NULL) if_label = malloc (8);

   if_label = strcpy (if_label, name ());         /* Generate  label                 */
     type = numconvert (ptr);                     /* Check & conv lt and rt types    */

    c_store_code (if_label);                      /* Output code if an error         */
                                                  /* hasn't been found               */
    c_store_code (":\n");

    return (type);

}
```

APPENDIX K

ROCK COMPILER — CODE GENERATION MODULE

```
/*********************************************************************
* PUBLIC DOMAIN SOFTWARE                                             *
*                                                                    *
* Name      :  Code Generation Module                                *
* File      :  Code_Gen.c                                            *
* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                     *
* Started   :  02/06/87                                              *
* Archived  :  04/10/87                                              *
* Modified  :  04/13/87  Code output to vdisk   EC                   *
**********************************************************************
* This file contains the following modules for the PHI compiler      *
*                                                                    *
*       C_Store_Code           C_Startup       C_Off_Insert          *
*       C_Ending               C_Printcode     C_Ztor                *
*       Acode                  C_Jmp           C_Start_Proc          *
*       C_I_Const              C_I_Form        C_End_Proc            *
*       C_I_Op                 C_Call_Proc                           *
*                                                                    *
* Algorithm :                                                        *
* This module contains the procedures necessary for code generation. *
* C_startup initializes the run_time file, & the semantic checker will*
* call the procedures as necessary. Note that "c_store_code" is a    *
* genaric generator which will spew any string given as an arg to the *
* output file.                                                       *
*                                                                    *
**********************************************************************
* Modified  :  04/13/87  Code output to vdisk, drive "d:"   EC       *
*********************************************************************/

/**************************** Externals ******************************/
#include <semcheck.h>
#include <string.h>
#include <fcntl.h>                                  /* For level 1 I/O           */

extern FLAG err_found;                              /* Error flag                */
extern long curr_addr;                              /* Current virtual address   */

/****************************** Globals ******************************/
char *code_buffer;                                  /* Buffer for output code    */
int buff_ptr = NULL;                                /* Ptr to chars in output buffer */


/****************************** C_Store_Code *************************/
   void
c_store_code (string)                               /* Put str into the output buffer */
   char *string;                                    /* String to be printed      */
{int ptr = NULL;                                    /* Ptr to the chars in input str */
```

```
    if (!err_found) {                                   /* Compute only if no error found */
    while (*(string + ptr) != NULL) {                   /* Copy string char by char       */
        *(code_buffer + buff_ptr) = *(string + ptr);
    ++ptr;
    ++buff_ptr;             }
            }
}

/****************************** C_Jmp *******************************/
    void
c_jmp (name)                                            /* Gen code to insert jump command*/
    char *name;
{
   c_store_code ("jmp ");
   c_store_code (name);
   c_store_code ("\n");
}
/**************************** C_Start_Proc ***************************/
    void
c_start_proc (name)                                     /* Output name for start of asm   */
                                                        /* language procedure             */
    char *name;
{
   c_store_code (name);
   c_store_code (":\n");
}

/**************************** C_End_Proc *****************************/
    void
c_end_proc (name)                                       /* Output name for ending an      */
                                                        /* assembly language procedure    */
    char *name;
{
   c_store_code ("call del_scope\n");
   c_store_code ("ret\n");
   c_store_code (name);
   c_store_code (":\n");
}

/**************************** C_Call_Proc ****************************/
    void
c_call_proc (name)                                      /* Output call for an assembly    */
                                                        /* language procedure             */
    char *name;
{
   c_store_code ("call ");
   c_store_code (name);
   c_store_code ("\n");
}

/**************************** C_I_Form *******************************/
    void
c_i_form (num)                                          /* Generate call to put integer   */
                                                        /* formal addr onto stack         */
    char *num;
{
   c_store_code ("mov cx,");
   c_store_code (num);
   c_store_code ("\n");
   c_store_code ("call i_formal\n");
}
```

```
/**************************** C_I_Const ******************************/
    void
c_i_const (name)                                    /* Output code for assigning an   */
                                                    /* integer constant               */
    char *name;
{
   c_store_code ("mov ax,");
   c_store_code (name);
   c_store_code ("\n");
   c_store_code ("call iputvalue\n");
}
/**************************** C_I_Op **********************************/
    void
c_i_op (op)                                         /* Output code for int arith ops  */
    optype op;                                      /* Type of operation              */
{extern void terror ();

   switch (op) {
       case (ADD) : c_call_proc ("iadd");
           break;
       case (SUB) : c_call_proc ("isub");
           break;
       case (DIVIDE) : c_call_proc ("idivn");
           break;
       case (MULT) : c_call_proc ("imult");
           break;
       default : return;
         }
}
/*********************** Startup *********************************/
    void
c_startup ()                                        /* Open and initialize files      */
{  code_buffer = getmem (SIZEBUFFER);               /* Initialize buffer              */
   c_store_code ( "extrn  initial : near\n");       /* Write utilities needed         */
   c_store_code ( "extrn  iadd    : near\n");
   c_store_code ( "extrn  isub    : near\n");
   c_store_code ( "extrn  imult   : near\n");
   c_store_code ( "extrn  idivn   : near\n");
   c_store_code ( "extrn  iequ : near\n");
   c_store_code ( "extrn  ineq : near\n");
   c_store_code ( "extrn  igt : near\n");
   c_store_code ( "extrn  ilt : near\n");
   c_store_code ( "extrn  land : near\n");
   c_store_code ( "extrn  lor  : near\n");
   c_store_code ( "extrn  igteq : near\n");
   c_store_code ( "extrn  iputvalue : near\n");
   c_store_code ( "extrn  ilteq : near\n");
   c_store_code ( "extrn  igetvalue : near\n");
   c_store_code ( "extrn  initial : near\n");
   c_store_code ( "extrn  finis : near\n");
   c_store_code ( "extrn  print_top : near\n");
   c_store_code ( "extrn  negation : near\n");
   c_store_code ( "extrn  i_formal : near\n");
   c_store_code ( "extrn  i_mov : near\n");
   c_store_code ( "extrn  ppush : near\n");
   c_store_code ( "extrn  ppop : near\n");
   c_store_code ( "extrn  add_scope : near\n");
   c_store_code ( "extrn  del_scope : near\n");
   c_store_code ( "org 0100h\n\n");
   c_store_code ( "cseg\n");
   c_store_code ( "call initial\n");
}
```

```
/************************ C_Print_Code ******************************/
   void
c_print_code ()                                   /* Output code buffer to          */
                                                  /* secondary storage              */
{extern char prefix [];
 int code;                                        /* Output file                    */
 char holder[30];

 strcpy (holder, "d:");                           /* set up file name               */
 strcat (holder, prefix);
 strcpy (prefix, holder);                         /* save prefix & drive for fut use*/
 strcat (holder, "a.86");

   code = open(FILENAME,O_TRUNC | O_WRONLY,NULL); /* Open file for writing and      */
                                                  /* overwriting only               */
   write (code, code_buffer, buff_ptr);           /* Write the buffer               */
   close (code);                                  /* Close the output file          */
}

/************************ C_ Ending *******************************/
   void
c_ending ()                                       /* Ending for output code         */
{
   if (!err_found) {
     c_store_code ("call print_top\n");
    /* Print address pointed to by   */
                                                  /* top of program stack           */
     c_store_code ("call finis\n");               /* Routine to make clean ending   */
     *(code_buffer + (buff_ptr ++)) = CNTRL_Z;    /* If no error, put asm language  */
                                                  /* delimiter to file              */
     c_print_code ();                             /* Output code to a file          */
                }
}

/*************************** c_ztor *******************************/
   void
c_ztor ()                                         /* Gen code for conv int to real  */
{}                                                /* Empty now, but watch our smoke */

/****************************** Acode *****************************/
   void
acode (ptr, type)                                 /* NOTE : USES EMPTY STATEMENTS   */
                                                  /* FOR REAL OPERATIONS            */
   nodal ptr;
   FLAG type;                                     /* Generate code for arith ops    */
{extern void terror ();
 int name;

   name = ptr->name;

   switch (name) {
    case (ADD_) : if (type == REAL);              /* Addition                       */
             else c_i_op (ADD);
             break;
    case (SUB_) : if (type == REAL);              /* Subtraction                    */
             else c_i_op (SUB);
             break;
    case (MULT_) : if (type == REAL);             /* Multiplication                 */
              else c_i_op (MULT);
              break;
    case (RDIV_) :                                /* Real Division                  */
              break;
    case (IDIV_) : c_i_op (DIVIDE);                    /* Integer Division                */
            break;
    }
}
```

APPENDIX L

ROCK COMPILER — USER INTERFACE

```
/*********************************************************************
* Name      :  User Interface                                        *
* File      :  User.C                                                *
* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                     *
* Started   :  04/01/87                                              *
* Archived  :  04/10/87                                              *
* Modified  :                                                        *
**********************************************************************
* This file contains the following modules for the PHI compiler      *
*                                                                    *
*         User_err            Getname              Prog_name         *
*         Print_header        P_Close              User              *
*                                                                    *
* Algorithm :                                                        *
*      This module contains the procedures necessary for the user in-*
* terface.                                                           *
*     Prog_Name gets the user's choice of program by calling Get_Name*
* Print header is called to print the initial screen display on con- *
* sole, & the User procedure is the overall coordinator of the inter-*
* face.                                                              *
*     User_Err and P_Close are both independent procedures.  User_Err*
* handles output in the event that an error or errors have been found.*
* P_close is called by "Rock_Main" to ensure the input file has been *
* closed.                                                            *
*                                                                    *
**********************************************************************
* Modified  :                                                        *
*********************************************************************/

/***************************** Externals ***************************/
#include <user.h>
#include <dos.h>                                  /* for "getch ()"          */
#include <stdio.h>

extern void clrscr (), mov_cursor (), clr_window ();

/****************************** Globals *****************************/

char u_name [BUFFLENGTH],                         /* Name of Source file     */
     prefix [BUFFLENGTH];                         /* Prefix of source file   */

FILE *infile;                                     /* File handle of source file    */

/***************************** User_Err *****************************/
   void
user_err ()                                       /* Screen interface for error msg */
{extern void clrscr ();
```

```
 extern int num_errors;                           /* Number of errors found         */
 FILE *errors;                                    /* Error File                     */
 int numblocks,                                   /* Number of blocks to read       */
     count = 0;                                   /* Generic loop  variable         */
 char *buffer = malloc (BSIZE),
     input;                                       /* Keypressed after pause         */

   errors = fopen (ERRORFILE,"a");
   fprintf(errors,
      "number of errors = %d\n",num_errors);
   putc ('$', errors);                            /* Put EOF marker to file         */
   fclose (errors);

   clrscr ();
   errors = fopen (ERRORFILE, "r");

   numblocks = fread(buffer,BLOCKSIZE,20,errors); /* Read error mgs from error files*/
                                                  /* BLOCKSIZE   will allow whole   */
                                                  /* file to be read at once        */
   while (*(buffer + count) != '$') {
      putchar (*(buffer + count));
      ++count;              }

   printf ("\n \n \n");                           /* Skip lines to give appearance  */
                                                  /* of user friendliness           */
   printf ("%s", PAUSE);                          /* Pause to give user a chance to */
                                                  /* comtemplate his errors         */
   input = getch ();                              /* Eat keyboard input after pause */

   fclose (errors);
   clrscr ();

   if (input == ESCAPE) exit (1);                 /* If user pressed escape,        */
                                                  /* exit the program               */
}

/*************************************** Getname ***********************************/
    void
getname ()                                        /* Returns the user's choice      */
                                                  /* of file to compile             */
{int ch,                                          /* Single input character         */
    count = 0;                                    /* Buffer pointer                 */

   do {                                           /* Loop, get file name ltr by ltr */
    if ((ch = getch ()) == BACKSPACE) {           /* <- key is hit                  */
        if (count) { --count;
            putchar (ch);                         /* Backspace                      */
            putchar (' ');                        /* Insert blank                   */
            putchar (ch);                         /* Eat last char if there is one  */
          }                   }

    else if (ch == ESCAPE) {                      /* Escape pressed; exit           */
        clrscr ();
        exit (1); }
    else if (ch < 127)
                        {                         /* Legitimate char read; use it   */
      putchar (ch);
      u_name [count] = ch;
      ++count;         }
     } while ((count <= BUFFLENGTH) &&
         ch != EOLN);                             /* Loop until buffer full or      */
                                                  /* return pressed                 */
   u_name [count - 1] = 0;                          /* Insert end of string char      */
}

/****************************** Prog_name ******************************/
    void
prog_name ()                                        /* Get legitimate program name    */
{
   do {                                             /* Loop until fopen finds         */
                                                    /* legit name                     */
     clr_window (9,1,21,79);                        /* Clear out lower window of scn  */
     mov_cursor (10,2);
     printf (GETPROGRAM);
     getname ();
     infile = fopen (u_name, "r");

     if (!infile) {                                 /* Name not in current directory  */
    mov_cursor (20,33);                             /* Print user friendly error msgs */
    printf (FILE1_ERROR);
    mov_cursor (21, 16);
    printf (FILE2_ERROR);

    if (getch ()  == ESCAPE) {                      /* Exit if ESCAPE pressed         */
            clrscr ();
        exit (1);
                   }
        }
      } while (!infile);                            /* Repeat until correct file found*/
                                                    /* NOTE - escape exits loop & prgm*/
   mov_cursor (13,28);
   printf (WAIT);
}
/**************************** Print_header ****************************/
    void
print_header ()                                     /* Print out header for user      */
{
   clrscr ();
   mov_cursor (1,33);
   printf (HEADER1);
   mov_cursor (2,24);
   printf (HEADER2);
}
/****************************** P_Close ******************************/
    void
p_close ()                                          /* Close out target file          */
{
   fclose (infile);
}
/******************************** User ********************************/
    void
user ()                                             /* Invoke user interface          */
{int count = 0;                                     /* Duty integer                   */
   print_header ();
   prog_name ();

   while (!(u_name [count] == '.'                   /* Copy root of input file name   */
        || u_name [count] == NULL)) {               /* Loop until end of input name   */
                                                    /* reached OR until end of str is */
    prefix [count] = u_name [count];                /* reached, if no  extension      */
    ++count;                  }
   prefix [count] = 0;                              /* Insert end of string value     */
}
```

APPENDIX M

ROCK COMPILER — RUNTIME UTILITIES

```
;*************************************************************************
;* Name      :  Phi Runtime Utilities                                   *
;* File      :  U.a86                                                   *
;* Authors   :  Maj E.J. COLE / Capt J.E. CONNELL                       *
;* Started   :  01/26/87                                                *
;* Archived  :  16 Feb 87                                               *
;* Modified  :  16 Apr 87 Stack/Varspace Crash error check  EC          *
;*************************************************************************
;
;*************************************************************************
;ALGORITHMS
;
; 1.   Input/Output:  The first section of the program contains input and output
;
; 2.   Virtual Space:  A virtual space is set up in the extra segment to hold both the
;    stack.  The middlej of this space is denoted by the symbol "vars", and variables
;    offset (± 32700) from vars.  In this implementation, the program stack grows from
;    vars grow from the bottom.  The virtual space is assumed to be made up of words
(two bytes), so only
;    even numbers may be used to access it.
;
; 3.   Stack:  The stack pointer is the si register, which is initialized to 32700.
;    grows, the si register is reduced by two. Ppush and ppop will push and pop two
;    registers. "Push_one" and "Pop_one" will push and pop  single words to and from
;
; 4.   Addressing Program  Variables: Each program variable is assigned a two-tuple A
;    scope and O is the offset from the base address of variables in that scope.
;    turn the address of a variable given A.
;
; 5.   Scoping:  Initially the scope is set to 0: the global scope.  The variable
;    space containing the outer scope, and the variable "S_Nest" contains the current
;    new scope is created, "S_Nest" is increased by one, and the three-tuple S =
;    (L = Static Link, pointing nesting level of the outer scope, N is the nesting
;    is the base address of display of variables for this scope.
;    When a scope is deleted, the top of the stack is saved, the top instantiation of S
;    and S_Link and S_Nest are recalculated.
;
; 6 Inserting/Extracting Program Variables: "I_Assign" will insert an integer or
;    scope contained in S_Nest when it is requested.  "Iputvalue" will insert the
;    resoponding tuple A on the stack.  "Igetvalue" will pop the tuple A off the top of
;    the value of the integer pointed to by A.
;
;*************************************************************************
;*************************************************************************
;* Modified  : 22 Feb 87 Add/del_scope changed to save TOS. EC           *
;*             16 Apr 87 Added check for stack/varspace crash, includes*
;*                       message to observer                            *
;*************************************************************************
```

```
;
;**************************************************************************************
;*                            Public Procedures                                        *
;**************************************************************************************
public i_mov
public i_formal
public igetvalue
public finis
public iputvalue
public find_addr
public add_scope
public del_scope
public initial
public finis
public ppush
public ppop
public iassign
public lor
public land
public iequ
public ineq
public ilt
public igt
public ilteq
public igteq
public negation
public iadd
public isub
public imult
public idivn
public print_top

;********************************************************************************
;*                                                                              *
;*                            I/O Procedures                                    *
;*                                                                              *
;********************************************************************************
;
;*********************** print_char ********************************
;Print a char to the screen
;assumes letter to be printed is in dl register
;
  print_char:
       push ax                                    ;save registers
       mov ah,06                                  ;put int vector
       int 21h
       pop ax
       ret

;************************* Eoln *********************************
;Prints end of line character to the screen
;
   eoln:   mov dl, 10                             ;Moves appro ascii values to crt
       call print_char                            ;IBM specific
       mov dl, 13
       call print_char
       ret

;********************* Print_Num *******************************
;Prints, as a number, the value found in the bx register
```

```
;
;
   print_num:     push ax
       push bx
       push cx
       push dx
       mov cx, 10000                              ;Base for dividing
       cmp bx,0                                   ;Check if negative
       jge small                                  ;If not, jump to start
              mov dx, '-'                         ;Emit negative sign
       call print_char
              neg bx                              ;Negate

      small:    cmp bx, 10                        ;test if less than 10
       jl final

      div_loop:  mov ax, bx                       ;Divide bx by cx
       xor dx, dx                                 ;Set up dx register
               div cx
       cmp ax, 0
       jne p_loop                                 ;If not zero, jump
       mov ax, cx                                 ;Otherwise, decr cx by factor of 10
       mov cx, 10
       xor dx, dx
       div cx
       mov cx, ax                                 ;Mov ax to cx and continue
       jmp div_loop

     p_loop:  mov ax, bx                          ;Main printing loop
       xor dx, dx                                 ;Set up dx register
       div cx                                     ;Divide
       mov bx, dx                                 ;Move remainder to bx
       add ax, 48                                 ;Add for ascii
       mov dx, ax                                 ;Print
       call print_char
       xor dx, dx                                 ;Set up dx for division
       mov ax, cx                                 ;Divide base value by 10
       mov cx, 10
       div cx
       mov cx, ax
       cmp ax,1                                   ;If base value 1, end loop
       jne p_loop                                 ;Else continue

     final:
       add bx, 48                                 ;Print final value
       mov dx, bx
       call print_char
       call eoln                                  ;End of line
       pop dx
       pop cx
       pop bx
       pop ax
       ret

;*********************** Print_top *********************************
;Prints the space pointed to by the top tuple of the program stack
;
   print_top:     mov di,si
       add di,2
       mov dx, vars[di]                           ;Get nesting level
       add di,2
       mov cx, vars[di]                           ;Mov offset to cx
```

```
        call find_addr                              ;Mov address into si reg
        mov di, cx
        mov bx, vars [di]                           ;Mov num from address to bx
        call print_num                              ;Print number
        call eoln                                   ;Inset eoln
        ret
;************************ print_s *****************************
;assumes address of is in the dx register
;assumes string ends with a "$" sign
;
  print_s:

        push ax                                     ;save register
        mov ah, 9
        int 21h
        pop ax
        ret

;*****************************************************************
;*                                                               *
;*                   Stack Procedures                            *
;*                                                               *
;*****************************************************************

;************************ Ppush **********************************
;Pushes values from cx (offset) and di (nesting level)
;
  ppush:   mov vars [si], cx                      ;Put offset in stack
        sub si, 2                                 ;Inc stack pointer
        mov vars [si], di                         ;Put Nest level into stack
        sub si, 2                                 ;Inc stack pointer
        cmp si, curr_addr                         ;Check for stack/varspace crash
        jg p_return                               ;If no crash, go to end
        mov dx, offset crash                      ;Get string for error message
        call print_s                              ;Print it
        call finis                                ;Halt execution

        p_return:      ret

;************************ Push_one ****************************
;Push a single integer from cx register to the program stack
;
  push_one:   mov vars [si], cx                     ;Put word in stack
        sub si, 2                                   ;Inc stack pointer
        ret

;************************* PPop ********************************
;Pop values from the program stack to di (nesting level) and cx (offset)
;
 ppop:     add si,2                                 ;Set up ptr
        mov di, vars [si]                           ;Get nesting level
        add si,2                                    ;Recalc pointer
        mov cx, vars [si]                           ;Get offset
        ret

;************************ Pop_One ******************************
;Pop a single integer from the stack to the cx register
;
  pop_one:    add si, 2                             ;Set up pointer
        mov cx, vars [si]                           ;Get word
        ret

;******************************************************************
;*                 Varspace Management Procedures                 *
;******************************************************************

;*********************** IAssign *********************************
;Assign an integer value to a variable space in current scope
;Assumes value is in ax; offset is set to current max offset
;
  iassign:    mov di, s_link
    ;get static link
        sub di,2                                    ;decrement it to pt to base address
        mov di, vars[di]                            ;mov base address to di
        add di, max_offset                          ;add offset
        mov vars[di], ax                            ;mov number into that address space
        add max_offset,2                            ;Inc max offset and current address
        add curr_addr,2
        ret

;************************ Igetvalue *****************************
;Pop the stack and move the integer value pointed to into the ax
register
;
    igetvalue:    call ppop;                        ;Get nesting level and offset
        mov dx, di
        call find_addr                              ;Get addr of (S_Nest, Max_Offset)
        mov di, cx
        mov ax, vars [di]                           ;Get integer value
        ret

;*********************** Iputvalue *****************************
;Takes an integer from AX register, puts its value into varspace,
;then puts its address on the top of the stack
;
   iputvalue:   mov dx, s_nest                      ;Get static nesting level
        mov cx, max_offset
        call find_addr                              ;Get addr of (S_Nest, Max_Offset)
        mov di,cx
        mov vars [di], ax                           ;Put value into memory
        mov di, s_nest
        mov cx, max_offset
        call ppush                                  ;Store (S_Nest, Max_Offset)
        add max_offset, 2                           ;Inc max offset and curr_addr
        add curr_addr, 2
        ret
;
;******************************************************************
;*                                                                *
;*                     Scoping Procedures                         *
;*                                                                *
;******************************************************************

;********************* Find_Addr *********************************
;Returns address of variable at nesting level dx, offset cx to cx reg
;
   find_addr: mov di, s_link                        ;Get addr of current static pointer

   find_loop: cmp es:vars[di],dx                      ;If stack value = scope, exit loop
        je f_out
```

```
       add di,2
       mov di, es:vars[di]                                   ;Else jump to next scope and loop
       jmp find_loop

    f_out:    sub di,2                                    ;Calc ptr to base addr of scope vars
       add cx, es:vars[di]                                                          ;Add offset
       ret
;******************** Add_Scope ***********************************
;Start new scope by adding static link, starting address, & nesting
level
;
   add_scope: mov cx, s_link                              ;Get static link
       inc s_nest
       mov di, s_nest                                     ;Get new nesting level
       call ppush                                         ;Save link and level
       mov cx, curr_addr
       mov di, max_offset
       call ppush                                         ;Save curr addr
       mov max_offset, 0                                  ;Re initialize max offset
       mov s_link, si
       add s_link,6
       ret

;******************** Del_Scope ***********************************
;Deletes a scope
;
   del_scope: call ppop;                                  ;Save top of stack
       mov dx, di
       call find_addr
       push cx                                            ;Save absolute address of tos
       dec s_nest                                         ;Reduce nesting level
       mov si, s_link                                     ;Decrease stkptr to current link
       sub si, 4
       mov cx, es:vars [si]
       mov max_offset, cx
       mov bx,2
       mov cx, es:vars [si+bx]
       mov curr_addr, cx
       add si, 6
       mov cx, es:vars [si]
       mov s_link, cx                                     ;Get current static link
       pop di
       mov ax, es:vars [di]                                           ;Restore top of stack
       call iputvalue
       ret

;*****************************************************************
;*                                                               *
;*                  Begin/End Procedures                         *
;*                                                               *
;*****************************************************************

;*********************** Initial **********************************
;initialize the stack and variables
;must initialize cx to base of stack heap before calling this
;
  initial:     mov si, SPACE_TOP                          ;Initialize base of stack
       mov di,0
       mov cx, 0
       call ppush                                         ;Push base_scope and address
       ret
```

```
;************************* finis ***********************************
;
  finis:
       mov ax,04c00h                              ;end procedure
       int 21h
       ret
;**********************************************************************
;*                            Booleans                                *
;**********************************************************************
;
;************************** Negation *********************************
;Negates a boolean value
;
  negation:   call igetvalue                      ;Get boolean
       cmp ax, 1
       jne zero
       mov ax,0
       jmp p                                      ;Jump to end
       zero:  mov ax,1
       p:  call iputvalue                         ;Stuff boolean & put addr on stack
       ret

;************************** Lor ************************************
;Takes logical or of two booleans and stacks address of answer
;
   lor:    call igetvalue                         ;get 1st boolean off stack to the cx
reg
       mov bx, ax                                 ;save boolean
       call igetvalue                             ;get 2nd value using the stack ptr
       or ax, bx                                  ;Perform or
       call iputvalue                             ;Put value into varspace & in stack
       ret
;
;************************* Land ************************************
;Takes logical and of two booleans and stacks address of answer
;
   land:  call igetvalue                          ;get 1st boolean off stack to cx reg
       mov bx, ax                                 ;save value
       call igetvalue                             ;get second value using stack ptr
       and ax, bx                                 ;Perform and
       call iputvalue                             ;Push boolean address onto stack
       ret
;
;************************* Iequ ************************************
;Takes logical equal of two integers and stacks address of answer
;
   iequ:      call igetvalue                      ;get 1st int off stack to the cx reg
       mov bx, ax                                 ;save value
       call igetvalue                             ;get 2nd value using the stack ptr
       cmp ax, bx
       je eql                                     ;Jump if equal
       mov ax, FALSE                           ;put false value into varspace
      cont:      call iputvalue                ;Put value into varspace, addr on stack
       ret
;
      eql:    mov ax, TRUE                     ;put true value into varspace
       jmp cont
       ret
;
```

```
;************************ Ineq ***********************************
;Takes logical not equal of two integers and stacks address of answer
;
   ineq:      call igetvalue                      ;get 1st int off stack to the cx reg
       mov bx, ax                                 ;save value
       call igetvalue                             ;get second value using stack ptr
       cmp ax, bx
       jne neql                                   ;Jump if equal
       mov ax, FALSE                          ;put false value into varspace
      fal:    call iputvalue                  ;Put value into varspace, addr on stack
       ret
;
      neql:   mov ax, TRUE                    ;put true value into varspace
       jmp fal
       ret
;
;************************* Ilt *********************************
;Takes logical less than of two integers and stacks address of answer
;Returns true if first value is less than the second value
;
   Ilt:   call igetvalue                          ;get 1st int off stack to the cx reg
       mov bx, ax                                 ;save value
       call igetvalue                             ;get 2nd value using the stack ptr
       cmp ax, bx                                 ;Compare
       jge less                                   ;Jump if less
       mov ax, TRUE                           ;put false value into varspace
      con:    call iputvalue                  ;Put value into varspace, addr on stack
       ret
;
      less:   mov ax, FALSE                   ;put true value into varspace
       jmp con
       ret
;
;************************* Igt *********************************
;Takes logical greater than of two integers and stacks address of answer
;Returns true if first value is greater than the second value
;
   Igt:   call igetvalue                          ;get 1st int off stack to the cx reg
       mov bx, ax                                 ;save value
       call igetvalue                             ;get second value using stack ptr
       cmp ax, bx                                 ;Compare
       jle greater_than                           ;Jump if greater than
       mov ax, TRUE                           ;put false value into varspace
      con1:   call iputvalue                  ;Put value into varspace, addr on stack
       ret
;
 greater_than:  mov ax, FALSE                 ;put true value into varspace
       jmp con1
       ret
;
;
;************************ Ilteq *********************************
;Takes logical ≤ of two integers and stacks address of answer
;Returns true if first value is less than or equal to the second value
;
  Ilteq:  call igetvalue                          ;get 1st int off stack to the cx reg
       mov bx, ax                                 ;save value
       call igetvalue                             ;get 2nd value using the stack ptr
       cmp ax, bx                                 ;Compare
       jg lteq                                    ;Jump if less to error
       mov ax, TRUE                               ;put false value into varspace
      con2:   call iputvalue                      ;Put value into varspace, addr on stack
       ret
;
        lteq:   mov ax, FALSE                     ;put true value into varspace
       jmp con2
       ret
;
;
;*********************** Igteq ********************************
;Takes logical ≥ of two integers and stacks address of answer
;Returns true if first value is greater than or equal to the second
value
;
  Igteq:   call igetvalue                            ;get 1st int off stack to the cx reg
       mov bx, ax                                    ;save value
       call igetvalue                                ;get second value using stack ptr
       cmp ax, bx                                    ;Compare
       jl  gteq                                      ;Jump if greater than or equal to
       mov ax, TRUE                               ;Pput false value into varspace
      con3:   call iputvalue                      ;Put value into varspace, addr on stack
       ret
;
        gteq:   mov ax, FALSE                     ;put true value into varspace
       jmp con3
       ret

;*****************************************************************
;*                      Integer Operations                       *
;*****************************************************************
;
;************************ Iadd **********************************
;Adds two integer values
;Assumes offset off second value is in SI register
;Offset of first value is at the top of the stack
;
   Iadd:   call igetvalue
       mov bx, ax
       call igetvalue                                ;First value to cx register
       add ax, bx                                    ;Perform addition
       jo err                                        ;if overflow, run time error
;
       call iputvalue                                ;Put integer into varspace
       ret
;
       err:   mov dx, offset add_err                 ;Error handler for overflow
       call print_s
       call eoln
       call finis
       ret

;************************ ISub *******************************
;Subs two integer values
;Assumes offset off second value is in SI register
;Offset of first value is at the top of the stack
;
   isub:   call igetvalue
       mov bx, ax
       call igetvalue                                ;First value to cx register
       sub ax, bx                                    ;Perform subtraction
       jo errs                                    ;if overflow, run time error
;
       call iputvalue                             ;Put integer into varspace
       ret
;
       errs:  mov dx, offset sub_err              ;Print error message on overflow
       call print_s
       call eoln
       call finis
       ret
;***************************** IMult ***************************************
;Multiplies two integer values
;Assumes offset off second value is in SI register
;Offset of first value is at the top of the stack
;
   imult:
       call igetvalue
       mov bx, ax
       call igetvalue                             ;First value to cx register
       imul bx                                    ;Perform mult, result in AX
       jc err1                                    ;if carry set, run time error
;
       call iputvalue                             ;Put integer into varspace
       ret
;
       err1:  mov dx, offset mul_err              ;put error message in dx register
       call print_s                               ;print it
       call eoln
       call finis                                 ;end
       ret

;***************************** IDivn ***************************************
;Divides two integer values, result in varspace, address of result
stacked
;Offset of first value is at the top of the stack
;
idivn:     push cx                                ;Save Registers
       push dx
       call igetvalue                             ;Get divisor
       mov bx, ax                                 ;Mov divisor to bx
           call igetvalue                         ;Get dividend to ax
;
       xor dx, dx                                 ;Set dx to 0
       mov cl,1                                   ;cl and ch are negative flags
       mov ch,1
       cmp bx,0
       jg test2                                   ;bx is positive, no action needed
       je errd                                    ;bx is 0, ERROR
       neg cl                                     ;bx is negative, cl flag negated
       neg bx                                     ;bx is made positive

   test2 :    cmp ax,0                            ;test dividend
       jge dloop                                  ;dividend >= 0, no action
       neg ch                                     ;ax is negative, ch flag negated
       neg ax                                     ;ax is made positive

   dloop:     sub ax,bx                           ;loop and count subtractions
       cmp ax,0
       jl  done                                   ;if ax less than 0, done
       inc dx                                     ;store result in dx
       jmp dloop                                  ;continue loop
   done:      mov al, cl                              ;Multiply ch and cl
      mul ch
         cmp al,0
      jge dend                                        ;if product not negative, no action
      neg dx                                          ;else negate answer
   dend:   mov ax,dx
      pop dx
      pop cx
            call iputvalue                            ;Put integer into varspace
      ret
;
   errd:      mov dx, offset div_err                  ;put error message in dx register
      call print_s                                    ;print it
      call eoln
      call finis                                      ;end
      ret

;*********************************************************************
;*                                                                   *
;*                 Function  Calling Procedures                      *
;*                                                                   *
;*********************************************************************
;
;***************************** i_mov *********************************
; Movs integer or boolean actuals with addresses at the top of stack to
; the lowest addresses within a scope
; Assumes bx has number of actuals needed to be moved

     i_mov:   pop ret_addr                            ;Save i_mov's return address
      call add_scope
     strt:   pop dx                                   ;mov addresses to cx and dxi regs
      pop cx
      call find_addr                                  ;Get virtual address of the integer
      mov di, cx
      mov ax, es:vars [di]                                       ;Set up ax for iassign
      call iassign
      dec bx
      cmp bx,0
      jne strt
      push ret_addr                                   ;Restore i_mov's return address
      ret

;******************** I_formal ***********************************
;Puts a formal to the top of the stack
;Assumes offset of formal in cx register

   i_formal:  mov di,0
      mov di, s_nest[di]                              ;Get nesting level
      call ppush                                      ;Push offset and nest onto stack
      ret

;*********************************************************************
;*                                                                   *
;*                          Variables                                *
;*                                                                   *
;*********************************************************************

dseg
```

```
;************************************* Constants *************************************

    TRUE           EQU    1
    FALSE          EQU    0
    SPACE_TOP           EQU     32700               ;Top of memory space

;******************************* Integer Variables *********************************

    max_offset        dw  0                            ;Maximum current offset w/in scope
    curr_addr         dw  -32700                       ;Current maximum address
        s_link        dw  SPACE_TOP                ;Current address of static link
    s_nest         dw  0                               ;Current static nesting level
        ret_addr      dw  0
;******************************** Error Messaages **********************************

       div_err        db  'DIVISION BY ZERO, FOOL!'
           db  '$'

    mul_err           db  'MULTIPLICATION OVERFLOW, IDIOT!'
           db  '$'

       add_err        db  'ADDITION OVERFLOW, DIMWIT!'
           db  '$'

    sub_err           db  'SUBTRACTION OVERFLOW, NITWIT!'
           db  '$'

        crash      db      'STACK/VARIABLE SPACE CRASH'
           db  '$'

;******************************** Error Messaages **********************************

eseg
    vars           dw  0
end
```

APPENDIX N - TEST SUITE

SIMPLE TESTS OF FUNCTIONS AND VARIABLES

```
let c : $Z -> $Z;
c (20) where c (n) == if 1 = 2 then 3 * n
            else 3 + n endif

--Simple "Hello I'm Alive Test"
```

```
let c : $Z -> $Z;
c (1 * 2) where c (n) == n * 3

-- Test for expression in functions's formals
```

```
let c : $Z -> $Z;
c (k + 2) where k == 2 and
        c (n) == if n = 1 then n * 3 else n + 4 endif

-- Test for expression in function's formals
```

TESTS FOR RECURSION

```
let c : $Z -> $Z;

c (k * 2) where k == 2 and c (n) == n * 3

-- Test for expression in function's formals
```

```
let c : $Z -> $Z;

c (0) where c (n) == if n = 0 then 1 else c (n - 1) * n endif

-- Test for recursion in functions
```

```
let c : $Z -> $Z;

c (5) where c (n) == if n = 0 then 1 else c (n - 1) * n endif

-- Test for recursion in functions
```

```
let c : $Z -> $Z;
```

```
c (3) where c (n) == if n = 0 then 1 else n * c (n - 1) endif

-- Test for recursion in functions


let c : $Z -> $Z;

c (7) where c (n) == if n = 0 then 1 else n * c (n - 1) endif

-- Test for recursion in functions
```

TESTS OF COMPLEX FUNCTIONS, INCLUDING BOOLEANS AS ARGUMENTS AND RESULTS

```
let c : $Z -> $B;

c (1) where
        c (n) == n = 6

-- Test for booleans in function


let c : $Z * $Z * $Z -> $Z;

c(2 - 1,3,4)  where  c(n,m,x) == n * m * x

--Test for multiple arguments


let c : $Z -> $B;
let d : $Z -> $Z;

c (1) where
    c (n) == 1 = d(1) where
        d(k) == k

-- Test for chaining in functions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $B;
c (3) where
    c (n) == 1 + d(n) where
        d(k) == if e(1)
        then k else k + 1 endif
          where e (k) == k = 3

-- Test for nesting in functions
```

```
let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $B;

c (3) * 10 where
    c (n) == 1 + d(n) where
         d(k) == if e(1)
         then k else k + 1 endif
           where e (k) == k = 3

-- Test for nesting in functions, result multiplied by constant
```

```
let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $B;

c (3) * c(4) where
    c (n) == 1 + d(n) where
         d(k) == if e(1)
         then k else k + 1 endif
           where e (k) == k = 3
         and b == 10

-- Test for two functions, same definition
-- Also, test for extraneous variable defined at end of program
```

```
let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $B -> $B;

c (3) * c(4) where
    c (n) == 1 + d(n) where
         d(k) == if e(2 = 3 ∧ 4 = 5)
         then k else k + 1 endif
           where e (k) == k

-- Test for boolean expression as an actual
```

TESTS FOR "AND" AND "WHERE" NESTING AND COMBINATIONS

```
let c : $Z -> $Z;
let d : $Z -> $Z;

c (3) * b where b == 10 and
         c (n) ==  n * d (n) where
           d (n) == 3

-- Test for nesting in functions
```

```
let c : $Z -> $Z;
let d : $Z -> $Z;

c (3) * b where b == 10 and
        c (n) ==  n * d (n) where
          d (n) == 3 * e where e == 10

-- Test for nesting in functions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c (3) + b where b == 10 and
    c (n) == d (1) + if n = e (1) then 2 else 10 endif
            where e (k) == -1 and
            d (g) == g + 5


-- Test for nested wheres and ands


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $B;

c (3) where
    c (n) == 1 + d(n) where
        d(k) == if e(1) then k else k + 1 endif
            where e (b) == b = 3

-- Test for nesting in functions



let c : $Z -> $Z;
let d : $Z;

c(5) where  c (n) == d
    and  d == 10  * 5

-- Test for single and statement
-- Test for datadef declaration


let c : $Z;
let d : $Z;
let e : $Z;

c where  c == (d  + 10 + e where e == 10)
```

```
   and  d == 10

-- Test for Multiple ands


let c : $Z;
let d : $Z;
let e : $Z;

c where  c == d  + 10 + e
   and  d == 10
   and  e == 10

-- Test for Multiple ands


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(5)  where  c(n) == d(n)  + 12
   and  d(s) == 10 + s

-- Test for Multiple ands using functions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(5)  where  c(n) == d(n)  + 12
   and  d(s) == 10 + e (s)
   and e(k) == 20 + k + t where t == 100

-- Test for Multiple ands , nested wheres


let c : $Z;
let d : $Z;
let e : $Z;

c where  c == d  + 10 + e where
    e == 10 and  d == 10
--Test for Multiple ands



let c : $Z -> $B;
let d : $Z -> $B;
let k : $Z -> $Z;
```

```
c(1) ∧ d(2)  where
    c (n) == n = 3 and
        d (n) == (1 = k (n - 1) where
          k (l) == l + 10)


-- Test for proper use of "and"  and  implementation of
-- Parens


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(5)  where  c(n) == d(n)  + 12 where k == 100
    and  d(s) == 10 + e (s)
    and e(k) == 20 + k

-- Test for Multiple ands, multiple wheres and formal/variable collisions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(5)  where  c(n) == d(n)  + 12 where k == 100
    and  d(s) == 10 + e (s) where t == 100
    and e(k) == 20 + k + t

-- Test for Multiple ands, multiple wheres and formal/variable collisions



let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(5)  where  c(n) == d(n)  + 12 where t == 100
    and  d(s) == 10 + e (s) where t == 120
    and e(k) == 20 + k + t

-- Test for Multiple ands, multiple wheres and formal/variable collisions
-- Also test to see if the proper "t" (120) was picked up



let c : $Z * $Z -> $Z;
let d : $Z * $Z -> $Z;
let e : $Z * $Z -> $Z;
```

```
c(5,1)  where  c(n,m) == d(n,m) + 12 where t == 100
   and  d(s,z) == 10 + e (s,z) where t == 120
   and e(k,l) == 20 + k + t + l

-- Test for Multiple ands, multiple wheres and formal/variable collisions
-- Test specifically for functions with multiple arguments



let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(5)  where  c(n) == d(n)  where t == 100
   and  d(s) == (e (s) where k == 2)
   and e(k) == 20 + t

-- Test for Multiple ands, multiple wheres and formal/variable collisions



let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(10)  where  c(n) == d(n)  where t == 100
   and  d(s) == e (s) where k == 10
   and e(r) == 20 + r + k

-- Test for Multiple ands, multiple wheres and formal/variable collisions



let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

c(10)  where  c(n) == d(n) + t where t == (r * 100 where r == 2)
   and  d(s) == e (s) where k == 10
   and e(r) == 20 + r + k

-- Test for Multiple ands, multiple wheres and formal/variable collisions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
let f : $N -> $Z;

c(10)  where  c(n) == d(n) + t where t == (r * 100 where r == 2)
   and  d(s) == e (s) where k == 10
   and e(r) == 20 + r + f (r)
```

```
    and f(r) == r

-- Test for Multiple ands, multiple wheres and formal/variable collisions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
let f : $N -> $Z;

c(10)  where  c(n) == d(n) + t where t == (r * 100 where r == 2)
    and  d(s) == e (s) where k == 10
    and e(r) == 20 + r + f (r)
    and f(r) == k

-- Test for Multiple ands, multiple wheres and formal/variable collisions


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
let f : $N -> $Z;

c(10)  where  c(n) == d(n) + t where t == (r * 100 where r == 2)
    and  d(s) == e (s) where k == 10
    and e(r) == 20 + r + f(r)
    and f(r) == if r = 0 then 100 else f (r - 1) endif

-- Test for Multiple ands, multiple wheres and formal/variable collisions
-- Test for if-then-else collisions with multiple ands, wheres


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
let f : $N -> $Z;
let zebra : $Z;

c(10)  where  c(n) == d(n) + t where t == (r * 100 where r == 2)
    and  d(s) == (e (s) where k == 10
    and e(r) == 20 + r + f(r) + zebra
    and f(r) == if r = 0 then 100 else f (r - 1) endif
    and zebra == t)

-- Test for Multiple ands, multiple wheres and formal/variable collisions
-- Test for if-then-else collisions with multiple ands, wheres


let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
```

```
c(5)  where  c(n) == d(n)  + 12 where t == 100
   and  d(s) == (10 + e (s) where k == 100
   and e(k) == 20 + k + t)

--Note the use of parenthesis here : if they are removed, the program will
--bomb because t will be undefined
```

ERROR TESTING

```
let x :$z;
let j:$Z;
let i:$z;

i where i ==x%j
    and x ==5 and j ==0

-- Gives Division by Zero run time error


let b:$b;
let i:$Z;
let j:$z;
let n:$n;
let x: $z;

if b then i
elsif ~(b ∧ b) then j
else x endif     where
    b == i=2 where
         i ==0
    and where j
    and where z == 69
-- Gives two parser errors : line 13 and 14, j undefined and
-- where following "and"

let fac : $N -> $N;

fac (5) where fac (n) == fac (n - 1)

-- Check for stack overflow


too_much where too_much == 1000 * 1000

-- Check for Multiplication Overflow


too_much where too_much == 30000 + 30000

-- Check for Addition overflow
```

```
too_much where too_much == -30000 - 30000

-- Check for Subtraction Overflow


let c : $Z -> $B;
let d : $Z -> $B;
let k : $Z -> $Z;
let g : $Z -> $Z;

c(1) ∧ d(2) where
          d (n) == (1 = k (n - 1) where
            k (l) == l + 10) and
               c (n) == n = 3

-- Test for proper use of comments; note that there is no
   delimiter on the second line of comments, as there should
-- be
```

MISCELLANEOUS TESTS

```
let b:$b;
let i:$Z;
let j:$z;
let n:$n;
let x: $z;

if (b ∨ ~b) then i
elsif (b ∨ ~b) then j
else x endif     where
    b == i=2 where
         i ==0
    and j ==2
    and x == 69
-- Test for not construct, boolean constructs


let b:$b;
let i:$Z;
let j:$z;
let n:$n;
let x: $z;

if ~(b ∨ ~b) then i
elsif ~(b ∧ ~b) then j
else x endif     where
    b == i=2 where
         i ==0
    and j ==2
    and x == 69
```

```
-- should give 2
-- Check and, or, notand, notor
-- Check if, else, elseif
-- Especially, check all in combination


let a:$Z;
let b:$z;
let y:$n;
let x: $z;
let f: $n*$n->$n;
let times : $n*$n->$n;

f(30,30) where
   f(a,b) == times(a,b) where
       times(x,y) == x*y
-- Multiargument Checking
-- Natural Type Checking


let a:$Z;
let b:$z;
let y:$z;
let x: $z;
let f: $z*$z->$z;
let times : $n*$n->$z;

f(30,4) where
   f(a,b) == times(a,b) where
       times(x,y) ==
         if ( 1 = 1) then x%y
              else 2 endif  end
-- Integer Division Checking


let c : $Z -> $B;
let d : $Z -> $B;
let k : $Z -> $Z;
let g : $Z -> $Z;

c(1)  ∧ d(2)  where
        d (n) == (1 = k (n - 1) where
             k (l) == l + 10) and
             c (n) == n = 3

-- Test for proper use of "and"  and  implementation of
-- Parens
```

APPENDIX O - ROCK COMPILER USER'S MANUAL

I. Installation

The rock compiler program comes on a 5.25" disk with all public domain programs necessary to run it. To install this program on another floppy disk or a hard disk, use the following procedures:

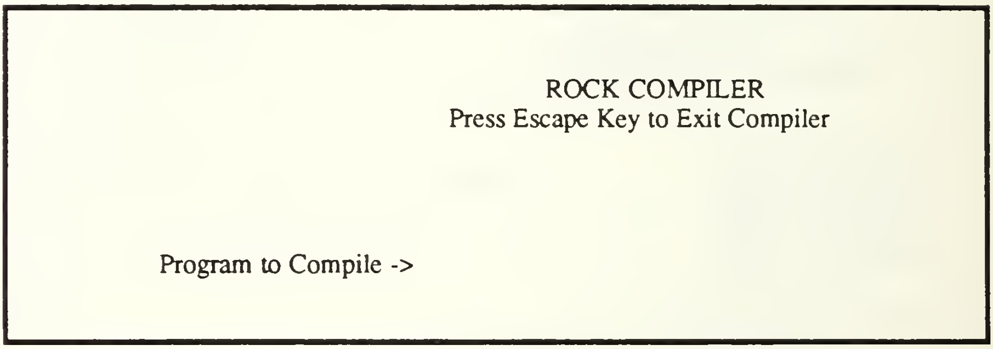
1) Change the system drive to the disk drive containing the floppy disk.

2) Type "INSTALL", followed by a space and the drive and directory on which you want the program installed.

Note that the Rock compiler uses three unsupplied files to operate: RASM86, LINK86, and your choice of word processor. The RASM86 and LINK86 programs must be installed on the same directory as the compiler.

II. Running the Compiler

a. Type in "ROCK" and wait for the screen display shown in figure 1 to appear.

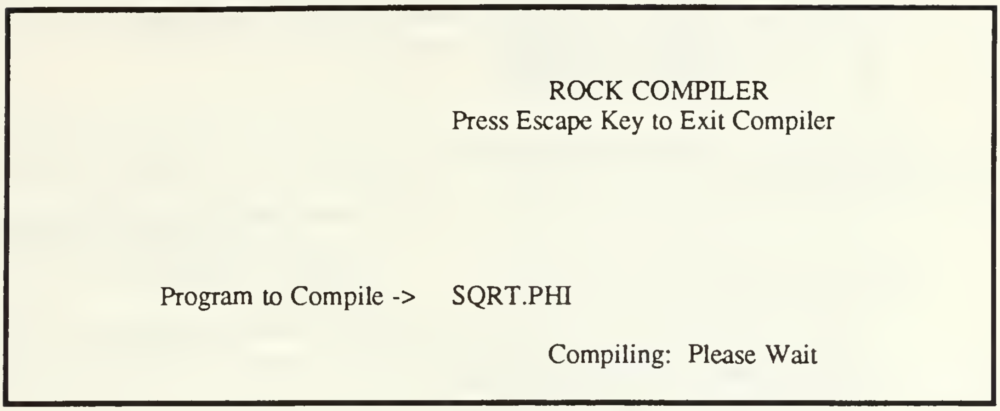


```
                              ROCK COMPILER
                       Press Escape Key to Exit Compiler



        Program to Compile ->
```

Figure 1

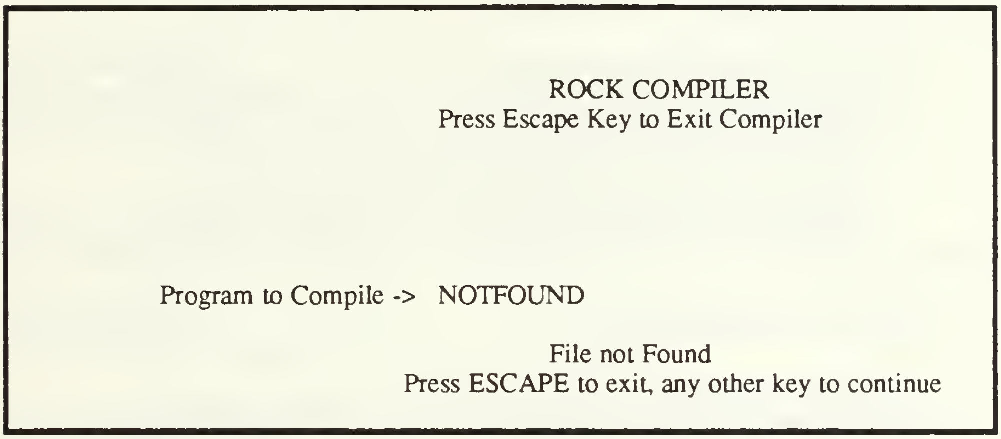
b. When the prompt appears, type in the file name of the source file you want to compile, then press return. The compiler will accept directory specifications in the file designation. If the source file is found, the compilation will begin immediately, and the screen will appear as shown in figure 2. If the file is not found, the screen will appear as shown in figure 3.

c. If a successful compilation takes place, the prompt for a source file reappears. If the compilation is not successful, error messages will appear on the screen, and a copy of these messages can be found in a file named Errors.Phi. A typical error display is shown in figure 4. After perusing the errors, you may press any key to return to the prompt for a source file.



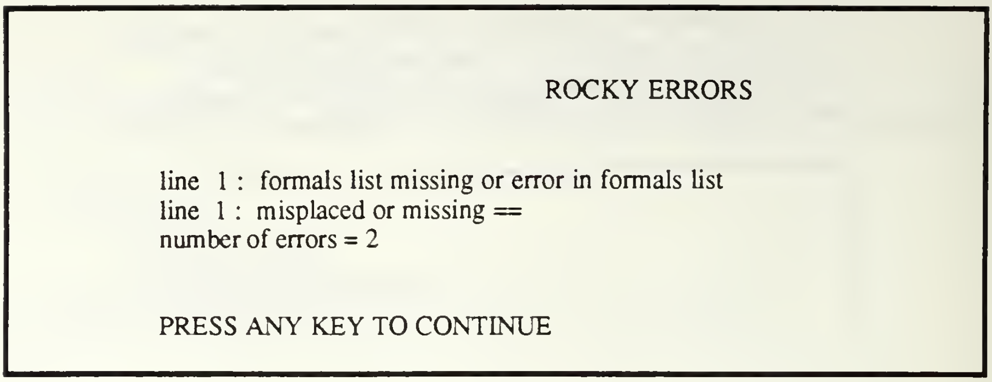
```
                              ROCK COMPILER
                        Press Escape Key to Exit Compiler



     Program to Compile ->   SQRT.PHI

                              Compiling:  Please Wait
```

Figure 2



```
                              ROCK COMPILER
                       Press Escape Key to Exit Compiler



     Program to Compile ->  NOTFOUND

                              File not Found
                    Press ESCAPE to exit, any other key to continue
```

Figure 3



```
                              ROCKY ERRORS

line  1 :  formals list missing or error in formals list
line  1 :  misplaced or missing ==
number of errors = 2


PRESS ANY KEY TO CONTINUE
```

Figure 4

d. If compilation is successful, both an .exe and an .obj file will be created. In the event that an error occurs, neither file will be created.
WARNING : If you choose to compile two programs with the same prefix, ensure you save the first one before compiling the second one; otherwise, the second compilation will overwrite the output file of the first compilation.

e. To cleanly stop the compiler, press the ESCAPE key any time the system asks for an input. If you have started to compile a program and you need a "panic" exit, press "Control-Break". If you do this, the cursor will not reappear on the screen. However, you can get it back by running the ROCK program again and making a normal exit.

III. Error Handling

Errors are divided into two categories : those found during compilation and those found during run time. The following two sections list the errors messages from both categories which you might encounter. Each message includes a brief synopsis of what causes the error.

COMPILER ERRORS

Message	Explanation
incomplete '|->'	Either an "|" or "|-" was found where "|->" was expected.
'\' without following '/', logical OR is '\/'	A single backslash was found where a logical or construct (\/) was expected.

'$' without following 'R','N','Z','B',or '1'	An incomplete type declaration was found.
invalid numeric constant ==> 3.	An illegal constant was found; in this example, "3."
literal without ending	An unterminated literal was found, or a literal spanned more than one line.
unidentified char in input program ==> #	A character with no meaning was found in the source file; '#', in this example.
MEMORY OVERFLOW DURING COMPILATION	The source program is too big for the host machine to compile.
error in statement following ==> *	An illegal statement follows the specified character; '*', in the example.
error in type definition following ==> *	An illegal type definition follows the specified character; '*', in the example.
unable to complete definition of blockbody after keyword LET	An unspecified error was found after LET, and the compiler is so completely sandbagged that it cannot recover.
missing or misplaced ';' after definition	A declaration, preceded by "LET", was not followed by a semicolon.
valid qualexp/exp not found in the def/auxdef	An invalid expression was found
valid typeexp not found in the def	An expression defining a type was either missing or incorrect.
formals list missing or error in formals list	Formals were expected but not found, or formals were incompletely specified.
misplaced or missing ')'	A PHI keyword or delimiter was expected or not found; ')' in the example.
at least one identifier must follow keyword TYPE	TYPE found without an identifier.
unable to complete def/auxdef following keyword AND	Improper or no expression found following AND.

missing or invalid auxdef after keyword WHERE	Improper or no definition following WHERE
missing or misplaced closing paren in formals list	Formals found without closing parenthesis.
error in processing multiple Actuals	One actual was found, but an error was spotted in a subsequent actual.
missing literal after keyword FILE	FILE was found without a filename being designated.
missing or invalid exp following KEYWORD	A keyword was spotted, but the following expression was illegal.
IF statement w/o ENDIF	No ENDIF to close off an IF statement.
error in formals preceding |->	"|->" found, but the formals list preceding it contained an error.
missing or invalid QualExp following COMMA operator	A list of elements was found with an illegal expression in it.
error in ArgBinding - check QualExp or closing bracket	An improper expression in an argument binding was found, or the closing bracket on an argument binding was not found.
OZONE LEVEL I -	Unimplemented feature found. for 19.99 the feature can be implemented in 1999
NUMERIC VALUE EXPECTED	Non-numeric type found where a numeric type was expected.
NATURAL EXPECTED	Natural type was not found where it was expected.
INTEGER OR NATURAL EXPECTED	Either an integer or natural type is proper, but neither was found.
ERROR IN TUPLE DEFINITION	A tuple is improperly defined : the source file used improper types or number of types in defining the tuple. This can also mean a single variable was improperly defined.
UNDEFINED VARIABLE IN AND SCOPE	An undefined variable was found in one of the two branches of an

	in its scope.
FUNCTION WITHOUT FUNCTION DEFINITION	A function was defined without a declaration of its type and formals.
FORMALS MISMATCHED	Formals in a function definition are not the same in either type or number as those in the function's declaration.
FUNCTION CALLED WITHOUT FUNCTION DEFINITION	No function definition found for the function called.
REAL NUMBER EXPECTED	An incorrect type was found where a real number was expected.
INVALID CONSTANT EXPRESSION	An invalid constant was found.
BOOLEAN VALUE EXPECTED	A boolean value was expected, but none was found.
BOOLEAN OPERATOR EXPECTED	A boolean operator was expected, but none was found.
OUT OF RUN-TIME MEMORY SPACE	Not enough space to accommodate the program during run-time.

RUN-TIME ERRORS

DIVISION BY ZERO	Division by zero attempted.
MULTIPLICATION OVERFLOW	A multiplication operation resulted in a numeric value outside the language limits.
ADDITION OVERFLOW	An addition operation resulted in a numeric value outside the language limits.
SUBTRACTION OVERFLOW	A subtraction operation resulted in a numeric value outside the language limits.
STACK/VARIABLE SPACE CRASH	The stack overwrote the variable space.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
2.	Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
3.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943	5
4.	D. L. Davis, Code 52Dv Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	B. J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
6.	Capt. J. E. Connell, USMC #5 Jack Ray Park Wentzville, Missouri 63385	3
7.	Maj. E. J. Cole, USMC 156 Haviland Rd Ridgefield, Connecticut 06877	3
8.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5002

Thesis
C53169 Cole
c.1 Implementation of a compiler for the functional programming language PHI - Φ.

Thesis
C53169 Cole
c.1 Implementation of a compiler for the functional programming language PHI - Φ.

thesC53169
Implementation of a compiler for the fun
3 2768 000 72987 5
DUDLEY KNOX LIBRARY